UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

Desarrollo de un motor gráfico sobre OpenGL para desarrollo de videojuegos 2D en dispositivos Android

Autor: Jorge Femenía del Rey
Tutor: Carlos Aguirre Maeso

MAYO-JUNIO 2014

Resumen

Hoy en día es difícil no advertir el gran uso que hace el ser humano de los dispositivos móviles inteligentes, o smartphones. De todas sus diversas aplicaciones, la que nos ocupa es la diversión en esos tiempos de espera que se producen en el día a día. Los desarrolladores de videojuegos han visto un mercado latente y muy productivo en esta combinación y las tiendas virtuales se han llenado con una gran diversidad de juegos que cumplen ese cometido.

Crear este tipo de juegos, aparentemente sencillos, debería ser un trabajo de creatividad. Sin embargo, muchos desarrolladores dejan atrás buenas ideas debido a la complejidad de los sistemas gráficos a los que hay que hacer frente. Para solucionar este problema, las opciones son escasas y en la mayoría de casos es necesario aprender a utilizar una herramienta, lo que cuesta tiempo, dinero y que finalmente limita sus posibilidades.

Este Trabajo pretende acabar con esas barreras y dar vía libre a desarrolladores independientes con conocimientos de programación, para que creen sus propios juegos 2D de forma rápida y sencilla, haciendo uso únicamente de su creatividad y sus conocimientos previos.

El documento expone el trabajo realizado para enmascarar las librerías gráficas subyacentes a todo juego, dejando tan solo una interfaz de alto nivel con múltiples posibilidades para la creación de juegos 2D en dispositivos Android, manteniendo en todo momento la máxima eficiencia en aspectos de rendimiento y la máxima simplicidad en aspectos de código.

Palabras Clave

SmartPhone, Android, OpenGL, Videojuego, 2D.

Abstract

Now a days is difficult not to observe the amount of use human beings make of their smatphone devices. From all their diverse applications, this particular case acknowledges the entertainment it produces on the waiting times of each day. Video game developers have found an increasing market, and a very productive one, in this combination, and the virtual stores have filled up with a great diversity of games to fulfill this mission.

Creating this kind of, apparently simple, games should be a creativity job. Instead, many developers left behind great ideas, because of the complexity of the graphic's systems they had to deal with. To solve this problem, they lack of options, and in most cases learning some pre-made tool is needed. This costs time and money, just to have their possibilities limited.

This Job intends to finish with those barriers, and clear the way for those independent developers with just programming knowledge. That way, they will be able to create their own 2D graphics games in a quick and easy way. Just making use of their creativity and their previous knowledge.

The document exposes the work done to mask the graphic libraries underlying every game, which will just leave a high level interface with multiple possibilities to create 2D games for Android devices. Maintaining, in every moment, the maximum efficiency when talking about performance, and the maximum simplicity when referring to coding.

KeyWords

SmartPhone, Android, OpenGL, Video game, 2D.

Índice de contenidos

Índice de Tablas

Índice de Figuras

Glosario

API: Interfaz de programación de aplicaciones, del ingles Application Programming Interface.

AVD: Emulador de Android, del ingles Android Virtual Device.

Downgrade: Degradado a un nivel inferior.

Frame: Fotograma o cuadro, es una imagen particular dentro de una sucesión de imágenes.

FPS: Fotogramas por segundo, del ingles Frames per second.

GPU: Unidad de procesamiento gráfico, del ingles Graphic Processing Unit.

HUD: Pantalla de visualización frontal, del ingles Heads Up Display.

IDE: Entorno de desarrollo integrado, del ingles Integrated Development Enviroment.

JNI: Interfaz Nativa de Java, del ingles Java Native Interface.

NDK: Kit de desarrollo nativo, del ingles Native Development Kit.

Pipeline: Proceso segmentado.

Render: Proceso de generar una imagen o video por calculo computacional.

SDK: Kit de desarrollo software, del ingles Software Development Kit.

Sprite: Pequeño mapa de bits que se dibuja en una pantalla.

Smartphone: Teléfono inteligente, construido sobre una plataforma informática móvil.

UPS: Actualizaciones por segundo, del ingles Updates per second.

1. Introducción

1.1. Motivación del proyecto

Desde el lanzamiento de *Pong* el 29 de Noviembre de 1972, la industria de los videojuegos ha crecido exponencialmente haciendo uso de todas las posibilidades de la nueva era informática. Hoy en día es una de las mayores industrias del mundo, genera ingresos millonarios y crea trabajo para multitud de personas.

Las consolas y el ordenador acaparaban la mayor atención del sector, pero con la aparición de los smartphones y las tablets se ha abierto un nuevo mercado y un acceso aun mayor al público, compitiendo directamente con consolas portátiles como la *Nintendo DS* o la *PSVita*. Los videojuegos han pasado a ser parte del día a día de muchas personas que antes no los utilizaban en absoluto. Además, los mercados de los smartphones son mundiales, creando un acceso global sin problemas de distribución.

Es por eso, que la idea de crear un videojuego para una plataforma móvil se torna muy interesante y éxitos como *Angry Birds* o *Fruit Ninja* (entre otros muchos), son pruebas fehacientes de ello. Son juegos sencillos, de bajo presupuesto que han batido récords de venta y esto llama mucho la atención.

Sin embargo, crear juegos para dispositivos smartphone no es fácil. A no ser, que trabajes para una compañía que tenga el trabajo ya hecho, las opciones para empezar de cero son escasas. Es por eso que crear una API gráfica con la cual trabajar para crear videojuegos 2D de forma rápida, sencilla y eficiente, se propone como una buena idea.

1.2. Objetivos del TFG

El objetivo de este TFG es la creación de un motor gráfico sobre OpenGL para dispositivos móviles Android. Esto es, una librería de funciones reutilizable que facilite el trabajo a desarrolladores, generando gráficos para videojuegos en dos dimensiones. La Librería debe ser capaz de, autónomamente, mover todos los gráficos en tiempo de juego, de la manera más eficiente y cómoda posible. La librería conforma una API, a la cual se referirá en este trabajo como API BTR, de las siglas Blue Tree Reality(nombre que se le ha asignado al proyecto), para evitar confusiones.

Para realizar el trabajo ha sido necesario, entre otras cosas, aprender el funcionamiento de la librería gráfica OpenGL y a raíz de este conocimiento hacer un diseño de clases superior que enmascare este trabajo, para hacerlo sencillo a futuros desarrolladores.

Además del motor gráfico, se han implementado una serie de librerías de físicas y de colisiones que dan más profundidad a la interacción del desarrollador de alto nivel con los gráficos generados.

1.3. Estructura de la memoria

Esta memoria, tras hacer una breve introducción al tema, comienza hablando del entorno de trabajo Android y de su uso en este trabajo. Luego explica los aspectos básicos de OpenGL, su funcionalidad interna y como se ha implementado en este proyecto.

Una vez se tiene una idea general del proceso de creación de gráficos se habla de la API desarrollada, de sus clases y de como ha sido pensada para que la utilice un desarrollador de alto nivel.

Finalmente, se termina explicando las implementaciones de un sistema de colisiones y otro de físicas para ayudar al programador a interactuar con la API creada.

Al final de este documento hay un manual de usuario[ANEXO 1]. Es muy recomendable seguir sus instrucciones y visualizar el código para acompañar las explicaciones.

2. Estado del arte y tecnologías a utilizar

2.1. ¿Qué motores gráficos existen?

Cuando hablamos de motores gráficos en realidad hablamos de motores de juego. Los motores gráficos son parte de los motores de juego y hacer una distinción solo serviría para acotar su cometido.

Antiguamente los videojuegos se programaban desde la base hardware. Los programadores tenían que lidiar con muchas variables que impedían la reutilización de código, incluso en sistemas más avanzados y en proyectos similares. Poco a poco y con el gran avance del Hardware, las compañías empezaron a utilizar motores de juego a un nivel muy bajo, cercano al Hardware, pero que adelantaban cierto trabajo. Los motores de juego de alto nivel, como el de este trabajo, aparecieron con la llegada del 3D.

A mediados de 1990, con los juegos *Doom* y *Quake*, salieron sus motores de juego que permitían a otros desolladores crear armas, personajes y niveles desde una base ya programada. Más tarde se han desarrollado cantidad de juegos con esta idea en mente, de manera que el propio juego pudiera crecer una vez lanzado al mercado, por personas ajenas a la empresa, por desarrolladores amateur, o incluso por los propios jugadores.

NeverWinter Nights es un ejemplo que va más allá. El famoso juego de Rol, apareció con la herramienta *Aurora Engine* que funcionaba sobre *DirectX*, competidor directo de *OpenGL*. Esta herramienta no solo fue utilizada para crear módulos para el propio juego sino, que años después la uso *CD Projekt*, un grupo independiente de programadores polacos, para crear *The Witcher*. La empresa tuvo tanto éxito que construyeron su propio motor y van por la tercera entrega de esta saga fantástica.

Figuras 1, 2 y 3: Logos de compañías mencionadas que codifican motores de juego.

Pero si hablamos de motores de juego, debemos mencionar *Unreal* [1]. Es quizás, el exponente más notable, ya que con sus librerías funcionan multitud de videojuegos de otros desarolladores. Ahora mismo existe una distribución portátil de este software que da un rendimiento nunca visto en móviles.

Para Android también existen motores de juego, el más conocido y usado es *Unity 3D* [2] que ha sacado recientemente una extensión para desarrollar en 2D directamente. Otro ejemplo digno de

[1] www.unrealengine.com

[2] www.unity3d.com

mención, es una librería llamada *Cocos2D*, con la cual se han hecho numerosos juegos para Android e iOS.

Sin embargo, la mayoría de motores mencionados en esa lista, requieren de un aprendizaje del propio motor y no requieren de un conocimiento vasto en informática. El motor gráfico de este trabajo pretende reducir el aprendizaje del motor al mínimo y a cambio espera conocimientos de programación por parte de los programadores que hagan uso de él.

2.2. ¿Con qué se ha realizado el trabajo?

El trabajo se ha realizado en eclipse versión Juno, utilizando el SDK de Android. Se ha utilizado el emulador AVD para realizar la mayoría de pruebas pero también se ha probado en varios dispositivos móviles, entre ellos: *Nexus 5*, *LG G2* mini y un *Sony Ericsson ARC S* .

La API BTR está dirigida a las versiones de la 8(*Foryo*) hasta la 18(*Jelly Bean MR2*) de Android. La 19(*KIT KAT*) no está implementada debido a que salió durante el desarrollo del proyecto, pero en principio no tendría problemas en funcionar en ella.

Para instalar un entorno de trabajo de forma rápida y sencilla Android provee su SDK con la IDE Eclipse preparada para usarse y está desarrollando un entorno de desarrollo propio llamado Android estudio, el cual no ha sido utilizado en este trabajo por estar en fase BETA.

Figuras 4 y 5: Logos de Eclipse versión Juno y Android.

3. Android

3.1. ¿Por qué Android?

A la hora de diseñar un motor gráfico para dispositivos móviles es muy importante determinar el sistema operativo al que se va a dar soporte. Los tres grandes del mercado son iOS, Windows Phone y Android. Este TFG enfoca su mirada sobre Android como único soporte.

Worldwide Smartphone Sales to End Users by Operating System in 2013 (Thousands of Units)				
Operating System	**2013 Units**	**2013 Market Share (%)**	**2012 Units**	**2012 Market Share (%)**
Android	758,719.9	78.4	451,621.0	66.4
iOS	150,785.9	15.6	130,133.2	19.1
Microsoft	30,842.9	3.2	16,940.7	2.5
BlackBerry	18,605.9	1.9	34,210.3	5.0
Other OS	8,821.2	0.9	47,203.0	6.9
Total	**967,775.8**	**100.0**	**680,108.2**	**100.0**

Tabla 1: Datos del mercado de smartphones por sistema operativo entre 2012 y 2013.

La elección de Android fue tomada, simple y llanamente por que el mercado es el más amplio. Las ventas de Android se han disparado desde su salida inicial y cada día tiene más adeptos. Es mucho más versátil y se puede usar en muchos más dispositivos.

Además, también aporta ventajas para desarrolladores. Android se programa sobre una versión de Java, un entorno de sobra conocido y es totalmente gratuito, desarrollar para Android no cuesta dinero. Es por esto que Android se convierte en una opción muy deseable. Su competidor directo, iOS, funciona sobre un lenguaje propio similar a C llamado *Objective C*, requiere de equipos específicos para programar y es necesario abonar anualmente una licencia de desarrollador para poder publicar cualquier aplicación en la store de Apple.

3.2. Preparación del proyecto

El trabajo consiste en crear un nuevo proyecto de aplicación Android. Dicho proyecto consta de una única actividad, en la cual estará el motor gráfico preparado y listo para usarse en forma de clases. El desarrollador deberá terminar la aplicación y formar el APK.

La librería creada ha tenido que lidiar con aspectos de Android como sus formatos de archivos, su flujo de trabajo o las resoluciones de sus diferentes dispositivos. Todo esto se irá mencionando a lo largo del trabajo en los puntos necesarios.

También se ha modificado el manifiesto para hacer cambios de estilo y comprobaciones de compatibilidad. Sin embargo, estos cambios no son estrictamente necesarios y será tarea del desarrollador estimar que necesita de acuerdo con la aplicación que vaya a diseñar. Por ejemplo, la orientación del dispositivo, la API BTR viene preparada para soportar las 4 posibles.

4. OpenGL ES 2.0

4.1. ¿Qué es el OpenGL?

Las siglas OpenGL significan, Open Graphic Library. Es una API que permite hacer uso de los recursos gráficos de diferentes plataformas para crear representaciones visuales en 3 dimensiones, proveyendo los comandos necesarios para renderizar escenas y presentarlas en pantalla.

También es multilenguaje, permitiendo programar en el lenguaje que más convenga para la aplicación que se quiera desarrollar y convirtiéndolo en multiplataforma. El conocimiento adquirido sobre esta API permite extender el trabajo a otros dispositivos, no solo a Android. Juegos tan conocidos como *Doom* o el tan popular ahora *Minecraft*, han sido creados a raíz de OpenGL.

OpenGL está disponible desde Android 2.0 a través del NDK usando Java. Esta implementado en iOS5, lo cual permitiría dar el paso a esa plataforma y también existe una distribución web, WebGL que admitiría expandirse a gráficos a través del navegador.

En cuanto al rendimiento, OpenGL hace uso de la GPU disponible en las tarjetas gráficas, reduciendo el uso del procesador. Esto es muy conveniente y es lo que ha permitido que juegos con una cantidad admirable de procesamiento, pudieran funcionar correctamente en dispositivos móviles.

En concreto en este TFG se utiliza la distribución OpenGL ES 2.0 del Grupo Khronos[Referencias, Recursos web 1.].

4.2. ¿Por qué OpenGL ES 2.0?

Entre las versiones portátiles que se ofrecen del OpenGL, llamadas ES del ingles sistemas embebidos, se ha elegido la 2.0 por dos motivos principales.

El primero, es el gran salto que se produjo de la versión 1.0 a la 2.0 en la cual se rompió con el modelo de programación conocido como fixed-funcion pipeline y se comenzó a utilizar uno nuevo, consistente en un nuevo sistema de "*shaders*" el cual se explicará más adelante, con el que han continuado sus predecesores hasta la fecha y con el cual planean seguir. Esto imposibilita hacer downgrades a versiones anteriores debido al cambio de sistema, pero una actualización a las últimas versiones es posible y sencilla.

Figura 6: Logo de las distribuciones portátiles de OpenGL.

La pregunta que surge a raíz de esto es ¿Por qué no utilizar la última versión(4.0)? Esto es por un tema de compatibilidad. Las últimas versiones de OpenGL ES necesitan de chips con arquitecturas específicas para poder ser utilizadas. Esto restringiría el mercado a unos pocos

teléfonos de última generación y para juegos de 2D no se iban a utilizar ninguna de las prestaciones que ofrecen estas nuevas actualizaciones.

4.3. ¿Cómo funciona el OpenGL ES 2.0?

En este punto se hace una breve introducción para tener una visión inicial que ayude a explicar el resto del trabajo. La siguiente imagen muestra el proceso segmentado que permite la representación gráfica a través de esta librería.

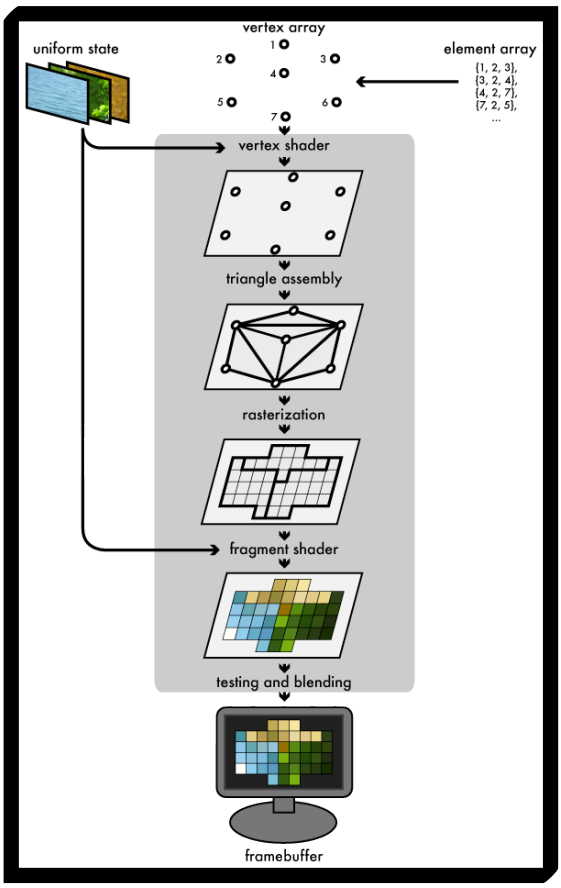


Figura 7: Diagrama del proceso de OpenGL. Se observa, la introducción de una cadena de vértices, como se transforma en una figura poligonal y a raíz de ella en un conjunto de pixeles. Finalmente se colorea y se muestra por pantalla.

Desde los inicios del 3D, sus objetos se han construido con triángulos. Millones de polígonos triangulares forman lo que hoy en día parecen objetos reales que se mueven con naturalidad. OpenGL no es una excepción.

Como se aprecia en la imagen, el programa que hace uso de OpenGL llena la memoria disponible con cadenas de vértices que representan posiciones. Estos vértices se proyectan en el espacio, son unidos por triángulos y luego pasan por un proceso, denominado rasterizado, que crea fragmentos del tamaño de pixeles a partir de la información aportada. A esos pixeles se les da un color y y se almacenan en el buffer de salida, que no es más que una imagen bidimensional de la escena tridimensional representada.

Todo este proceso se irá viendo a través de su implementación en este documento.

4.4. Configuración del OpenGL

Para usar OpenGL ES en Android, se ha creado una actividad que contiene una vista OpenGL con que representarán las imágenes renderizadas por la librería gráfica. La vista se adecua a la pantalla rellenándola totalmente y actuará de ventana al mundo 3D generado.

Por supuesto, el programa comprueba que el dispositivo tenga una versión de OpenGL 2.0 o superior y en caso de no tenerla lanza un mensaje de error para que el usuario sepa cual fue el problema. Aun así, el manifiesto de la aplicación se asegura de que esa comprobación ya la haga el usuario al bajar de la Store la aplicación, informándole de cualquier problema con la versión o el dispositivo al que se quiere descargar.

Una vez creado, OpenGL se ha configurado de la siguiente forma:

- Se ha marcado la versión como OpenGL 2.0
- Se ha seleccionado la clase de renderizado deseada, esta clase ya forma parte de la librería implementada en este trabajo y se verá en el siguiente apartado.
- Se ha configurado la frecuencia de dibujo. En este caso el control lo tiene la API BTR para poder llegar a la cantidad de FPS optima. Esto quiere decir, que será la API BTR la que haga una petición para que OpenGL dibuje.

Todos estos ajustes tienen lugar en la clase *MainActivity*, clase contenedora del método principal de la actividad de la aplicación. Concretamente en el método *onCreate*, que es el método al que llama Android cuando inicia la actividad y por lo tanto el método idóneo para hacer todos los ajustes previos a su uso.

4.5. La clase de renderizado

Es la encargada de plasmar en la pantalla la escena de dos o tres dimensiones que se quiera representar. Consta de 3 métodos, estos se ejecutan en el hilo perteneciente a OpenGL en los tiempos en los que el sistema Android lo estime oportuno, estos son:

- **onSurfaceCreated**: Se llama cuando se crea por primera vez, o si se pierde y se quiere recuperar. En este método se reservan todos los recursos que va a utilizar la escena y si el desarrollador no estima necesario cambiarlo es donde realizará la API BTR el trabajo. En el se cargarán las texturas y se añadirán a los sprites, de los cuales hablaremos más adelante.

> *Nota: En caso de desear realizar la carga de las texturas fuera de este método se ha implementado una ayuda para que las cargue desde otra pantalla en el tiempo de dibujo. Esto es necesario ya que solo se puede acceder al hilo de OpenGL a través de uno de estos métodos. Será mencionado más adelante cuando se hable de la API BTR y de los hilos.*

- **onSurfaceChanged**: Se llama cuando cambia la superficie sobre la que trabajamos, por ejemplo si el usuario cambia de modo apaisado a normal. En este método se establecen las dimensiones de la pantalla fijando la proporción adecuada entre la anchura y la altura para que la imagen quede proporcional a dichos espacios.

Nota: A pesar de que esté implementado correctamente y que funcione, el desarrollador tiene que tener cuidado con este método puesto que al cambiar de modo apaisado a normal, o viceversa, Android rehace la aplicación borrando todo contenido que no se haya almacenado, lo cual puede dar lugar a quebraderos de cabeza. Lo único que tiene que hacer el desarrollador es almacenar la información por algún medio de los que ofrece Android para dicho cometido [1], por ejemplo, preferencias.

- **onDrawFrame**: Cada vez que se dibuja un nuevo frame o fotograma se llama a este método. En él se ha especificado que objetos se van a dibujar y en que orden.

Estas clases han sido preparada en este TFG para ser autónomas, el desarrollador no tiene que modificar nada de estas clases, de ello se encarga la API BTR.

4.6. Los Objetos

Una vez el OpenGL está preparado, es necesario aprender a utilizarlo para dibujar. La librería, nos permite trabajar con tres clases de objetos, lineas, puntos y triángulos. Para crear un juego en 2D lo que necesitamos son sprites, esto son, cuadrados rellenos con texturas dinámicas o estáticas que representan un objeto bidimensional en el marco de la pantalla.

Se ha mencionado antes que el 3D se crea a partir de triángulos. Las muchas propiedades de los triángulos, entre ellas, que cualquier cuerpo geométrico puede ser descompuesto en triángulos, y estos a su vez también pueden ser descompuestos, les hacen fácil de manejar. Pero la verdadera razón por la cual se utilizan triángulos es por su eficiencia, el algoritmo de rasterización de triángulos es el más rápido y es nativo. Es por eso, que en este proyecto lo que se ha hecho es unir 2 triángulos, coincidentes en 2 vértices, para crear el susodicho cuadrado que representará el sprite.

Figura 8: Creación de un sprite rectangular a partir de dos triángulos.

OpenGL trabaja con arrays de información, de echo toda la información de la pantalla la introduce en buffers que luego se liberan dando lugar a la representación visual deseada, es por eso que la información que le damos a OpenGL para que pinte también va en dichos formatos. Esta es la forma en la que informamos a OpenGL de las dimensiones del cuadrado que vamos a dibujar. Le aportamos 6 vértices, los cuales representan los dos triángulos que formarán el cuadrado.

Todo este proceso lo realiza la clase TexturedQuad, en ella se definen las dimensiones del rectángulo y esta hace los cálculos para crear los vértices necesarios generando un rectángulo de esas dimensiones.

[1] http://developer.android.com/guide/topics/data/data-storage.html

4.7. Dalvik y la comunicación con OpenGL

La operación necesaria para pasarle la información de vértices a OpenGL se explica a continuación.

Debido a que el sistema Android funciona sobre diferentes dispositivos con diferentes arquitecturas, cuando se ejecuta la aplicación sobre un dispositivo, o sobre un emulador, no funciona directamente sobre este, como no lo hace Java, sino que lo hace sobre una máquina virtual llamada Dalvik. Esta máquina virtual no tiene acceso directo al Hardware sino que debe de hacer uso de APIs para llegar a él. OpenGL, por otro lado, opera sobre la GPU así que debemos aportarle los datos directamente para que esta pueda cumplir sus funciones.

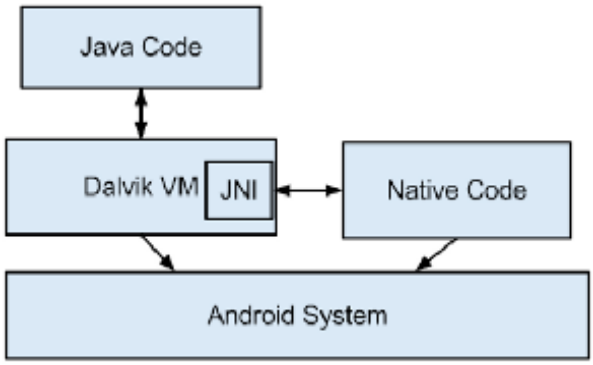


Figura 9: Esquema de capas de código Android. El código Java accede a la máquina virtual Dalvik que da ordenes al sistema o lo convierte en código nativo.

Como el código creado se ejecuta sobre Dalvik y OpenGL funciona directamente sobre la GPU, es necesario asegurarse que el tamaño de bytes utilizado es el correcto. La librería utilizada, *android.opengl.GLES20,* solventa este problema y funciona sobre JNI, la interfaz nativa de Java. Lo único que es necesario hacer, es reservar memoria nativa e introducir nuestros datos en ella de forma correcta. Para ello, se utilizan los buffers anteriormente mencionados, formados por floats, teniendo en cuenta que cada float son 4 bytes, 32 bits de precisión.

Nota: Dalvik tiene un recolector de basura y decide cuando liberar memoria, es por eso que ciertas funciones como los enums, no deben ser utilizados puesto que su implementación hace que se generen allocs continuamente, aumentando el tiempo de uso de la CPU. Aun así, esto nos afecta en esta sección puesto que el sistema nativo no funciona de la misma forma y no espera que se muevan bloques de memoria o que se eliminen de forma automática. Las texturas y datos que use OpenGL se almacenarán directamente en la RAM de la gráfica del dispositivo y OpenGL trabajará con direcciones de memoria fijas, las cuales solo se liberaran si se ordena de forma específica. Entonces si, Dalvik hará uso del recolector de basura cuando estime oportuno.

En la figura 10 se muestra un ejemplo para el caso de los vértices.

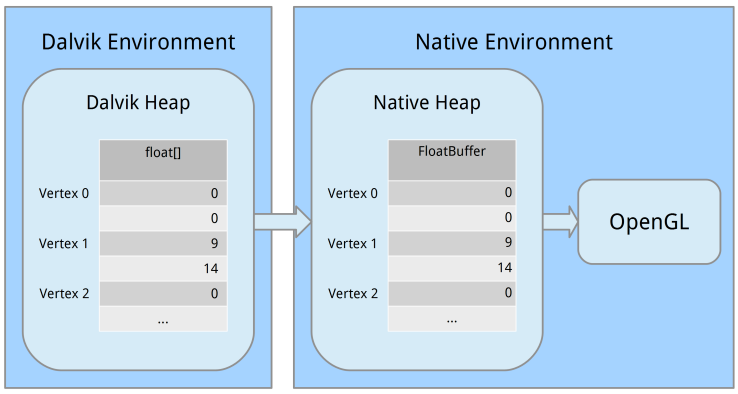


Figura 10: Conversión de datos de la pila de Dalvik a OpenGL. A un lado está la pila de datos que recibe Dalvik y al otro la pila nativa del JNI a la que accede OpenGL.

Finalmente, una vez tenemos la información de los vértices en la memoria, necesitamos enviársela a flujo de datos del OpenGl, para ello se utilizan subrutinas llamadas shaders.

4.8. Los Shaders

Los shaders son programas independientes utilizados por OpenGL con ciertos procesos intermedios en la creación de la escena final. Están escritos en un lenguaje similar a C denominado GLSL [1] de las siglas en ingles Graphic Library Shading Language, cuya extensión es esa misma, ".glsl".

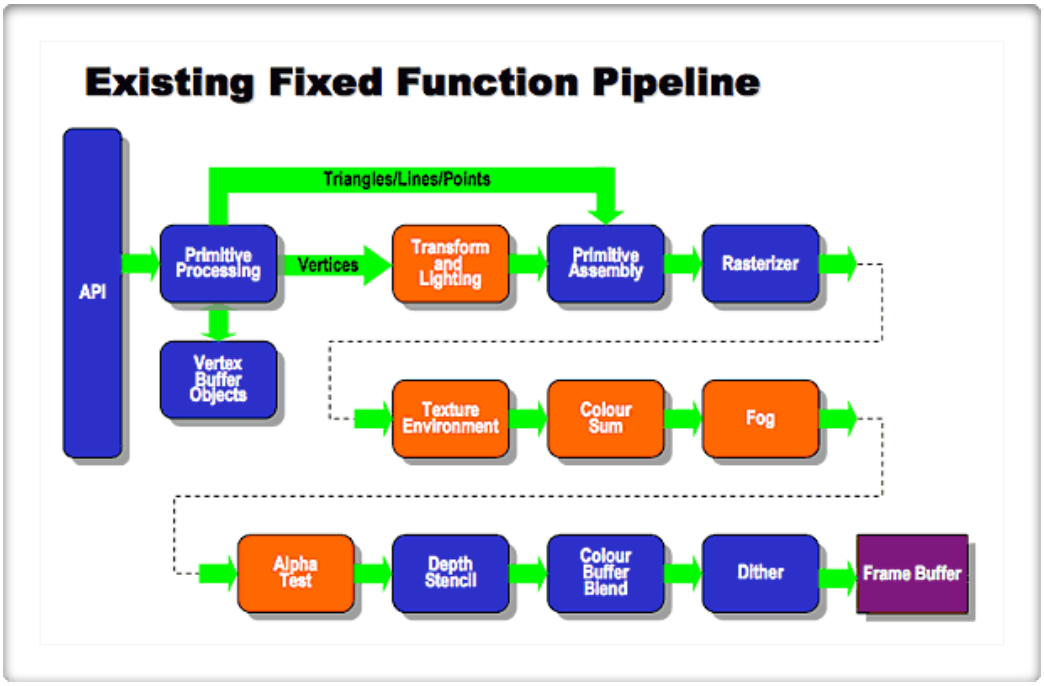


Figura 11: Antiguo Modelo de OpenGL. En el se marcan en naranja los procesos de los que ahora se encargan las shaders.

[1] http://www.khronos.org/files/opengles_shading_language.pdf

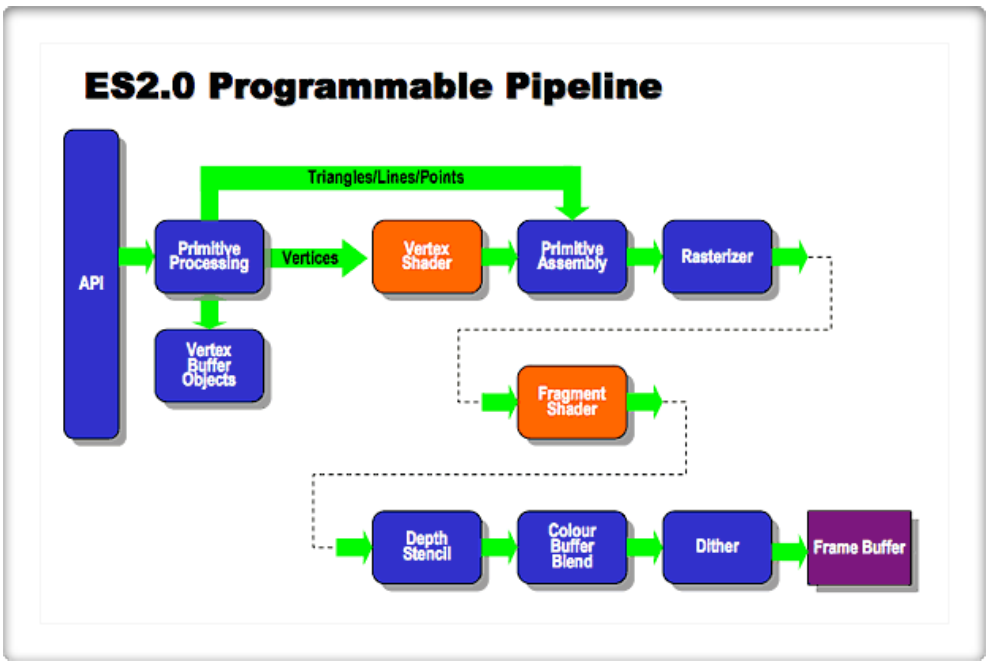


Figura 12: Nuevo Modelo de OpenGL. Las shaders están marcadas en naranja.

Como se puede ver en los diagramas, estos programas sustituyen funciones que antes hacia el flujo de trabajo directamente. Estas funciones eran fijas y no podían ser más que configuradas. Con el sistema de shaders dichas funciones pueden ser programadas, dando acceso a un mayor numero de posibilidades. Hay dos clases de shaders, Vertex Shaders y Fragment Shaders, de vértices y de fragmentos respectivamente:

- **Vertex shader**: Calcula la posición proyectada de los vértices dados en la superficie de la pantalla. También calcula la posición final de las coordenadas de las texturas.

- **Fragment shader**: Definen el color final de los objetos predefinidos por los vértices y su profundidad. En el caso de utilizar texturas, las aplica sobre el objeto.

Para este TFG se han implementado 3 tipos distintos de shaders, 1 por cada objeto diferente(animados, estáticos y batched). Estos objetos serán explicados más adelante.

4.8.1. Funcionamiento de los shaders

Los shaders utilizan 3 tipos distintos de parámetros de entrada y salida:

- **Uniformes**(*uniform*): Valores de solo lectura que no cambian durante el mismo render. Un ejemplo sería la posición de la cámara, que viene especificada a través de una matriz. Estos valores se encuentran en ambas clases de shaders.

- **Atributos**(*attribute*): Estos parámetros solo están disponibles en la clase vertex shader. Son también de solo lectura, pero en este caso varían por cada vértice. Un ejemplo sería la propia posición de cada vértice.

- **Variables**(*varying*): Por último, las variables se utilizan para pasar información de la clase vextex shader la clase fragment shader, el único requisito es que tengan la misma declaración en ambas.

Son de solo lectura en la clase de fragmentos pero se pueden modificar en la clase de vértices. Un ejemplo sería las coordenadas finales de las texturas.

Igual que con la mayoría de código, las shaders hay que compilarlas para obtener un programa que podamos utilizar. Este proceso lo hace la librería implementada al crear un objeto, no se compilan previamente con el código Java.

Para ello, primero carga el código de la shader. Hay dos formas de realizar esta tarea. En este TFG se ha decidido que lo mejor era crear los archivos *.glsl* representativos de las diferentes shaders de las que se va a hacer uso y luego cargarlas en memoria leyendo estos archivos.

Otra forma es escribirlas directamente en el código en un String, esto evita que sea necesario cargar el archivo pero es menos modular. Por eso, se creo la clase *TextReader* que simplemente lee un fichero de texto y lo almacena en memoria como un String. Esta clase obtiene los shaders de la carpeta *raw* almacenada en la carpeta *res*, carpeta de recursos habitual en todos los proyectos Android.

Una decisión que se tomo a la hora de implementar el programa es escribir en el log el proceso de compilación de las shaders. Esto es muy útil cuando se quiere modificar una shader, ya que podemos ver donde falló el proceso e ir directamente a donde se produjo el error.

Nota: Hay que tener en cuenta que aunque las funciones de compilación de las shaders devuelvan errores, son los fabricantes de los dispositivos los encargados de implementarlos, es por eso que muchas veces nos puede devolver error sin la especificación sobre que tipo de fallo ocurrió. También es necesario mencionar que el emulador funciona sobre un OpenGL de ordenador y hay sutiles diferencias, como por ejemplo, que en el emulador se pueden operar *floats* con *ints*, mientras que en un dispositivo real no. Es muy importante tener esto en cuenta y siempre utilizar un dispositivo móvil para comprobar el correcto funcionamiento de las shaders.

A la hora de crear el programa de shaders, es necesario que la librería implementada compile primero la clase vertex shader y luego la clase fragment shader, uniéndolas a posteriori. Esto se realiza en la clase *ShaderHelper*, que a raíz de los dos códigos necesarios de shaders devuelve la ID del programa creado.

Finalmente cuando tenemos compiladas las shaders y unidas en un programa, podemos utilizar este en nuestra clase renderer para dibujar un objeto. Para ello se enlazan los parámetros que usara el programa, tanto a vértices como a fragmentos. Esto se realiza, dando los mismos nombres que tenían en el documento GLSL y enlazándolos a su valor, en este caso una variable. La clase *ShaderProgram* realiza el proceso a un nivel básico, los hijos que extienden esta clase van aumentando su complejidad y especificidad según el tipo de objeto que van a crear.

Las shaders creadas para que use la API BTR son shaders básicas. No tienen efectos, simplemente colocan los vértices y las texturas según el tipo de objeto que sean. A parte de en el código, se pueden revisar en el [Anexo 2].

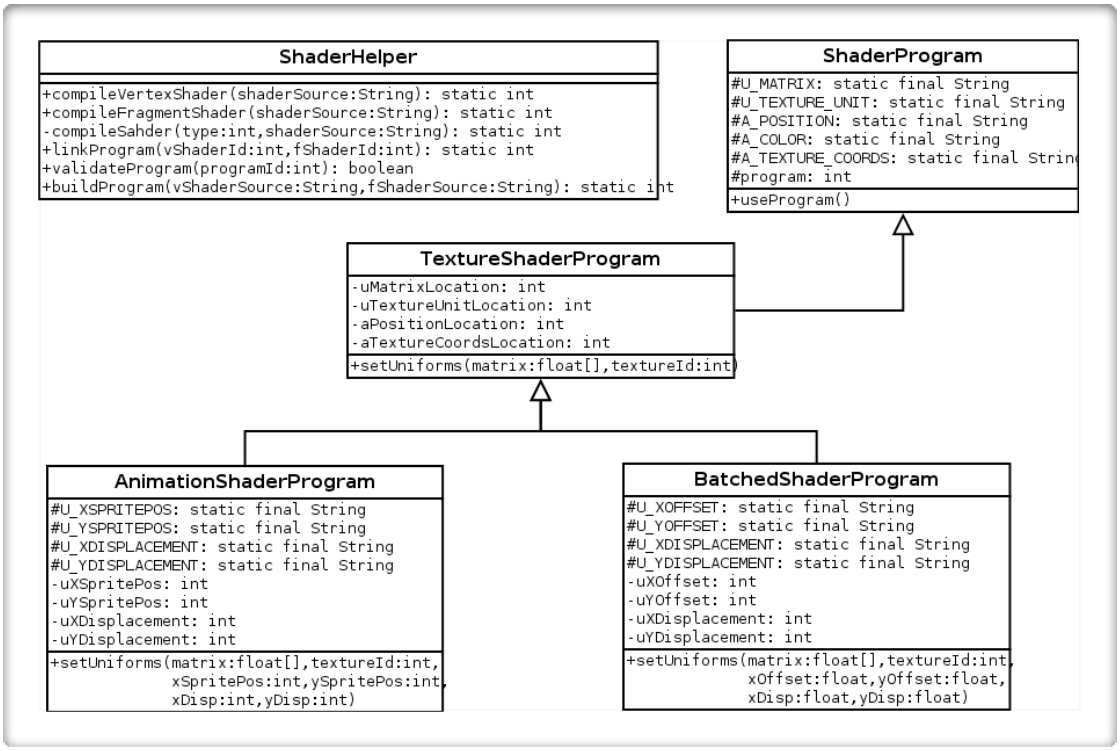


Figura 13: Diagrama de las clases que forman los programas de las shaders.

De nuevo, esta parte de OpenGL está cerrada a los desarrolladores de más alto nivel que no deben preocuparse de este proceso. Las clases del diagrama de la figura 13 empaquetan este proceso comprimiéndolo en una sola referencia. Es cierto, que el acceso a esta parte del programa daría mayor versatilidad de efectos, sin embargo, dándole acceso la complejidad de la API BTR aumentaría en gran cantidad. El programador tendría que adquirir conocimientos en este área y todo para efectos que en 2D se pueden crear mediante las texturas.

4.9. Dibujar

Para terminar el proceso y finalmente dibujar un objeto, lo que se hace es asignar el programa que se va a utilizar, como hemos visto, vertex shader y fragment shader unidas, y enlazar las entradas. OpenGL se ocupa de coger los datos y realizar el resto del proceso.

Se dibujan todos y cada uno de los objetos pedidos, dando lugar a una escena. Dicha escena se almacena en el buffer de frames y está lista para ser liberada directamente a pantalla cuando sea necesario.

4.10. Texturas

Una textura es un mapa de bits que cubre un objeto virtual como si fuera su piel. La forma de representar dibujos de dos dimensiones en OpenGL será proyectarlos sobre superficies cuadrangulares planas.

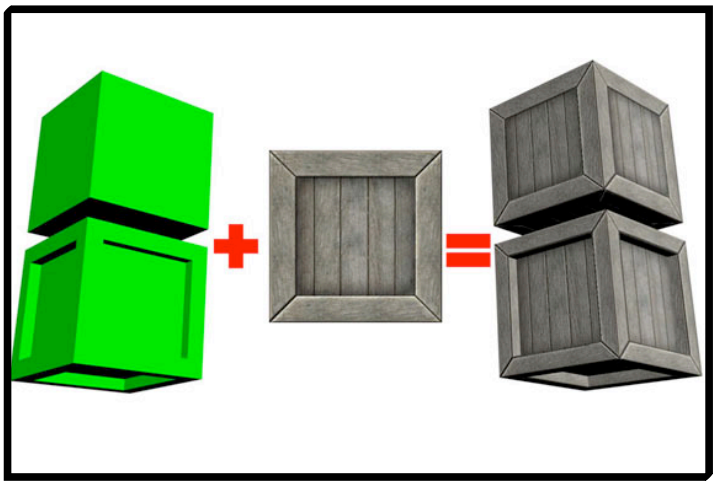


Figura 14: Ejemplo de aplicar una textura a un objeto en 3D.

Sobre los cuadrados que se crean, tal y como se explica en los apartados anteriores, se aplica la textura con el dibujo. A continuación veremos como se cargan las texturas y sus complejidades, y posteriormente veremos como se animan dichas texturas para crear sensación de movimiento.

Para cargar las texturas primero se carga una imagen a memoria que contenga la información gráfica. Esta imagen puede estar en formato JPG o PNG. Durante las pruebas realizadas en el trabajo se ha utilizado PNG que tiene menos compresión que JPG y da una resolución más alta para temas de dibujo lineal. Además, tiene canal alfa, muy importante para este tipo de aplicaciones, ya que nos permite definir la opacidad de los pixeles permitiéndonos la transparencia total. No querríamos tener cuadrados con dibujos pegados sobre fondos blancos.

A parte del formato las texturas no tienen más restricciones que su tamaño. El tamaño de la imagen debe ser potencia de 2, tanto el ancho como el largo, por ejemplo, 32x64 sería un formato válido. Esto es necesario, puesto que la textura se carga directamente en memoria y algunas arquitecturas no son capaces de trabajar con ciertos tamaños, alterando información y haciendo la imagen quede incompleta o equívoca. Esto es completamente lógico, como vimos en adicción de vértices al buffer de entrada de OpenGL, es necesario conservar ciertas dimensiones para que se trabaje con los mismos tamaños con los que lo hace el JNI. Aun así, algunas arquitecturas soportan otros tamaños.

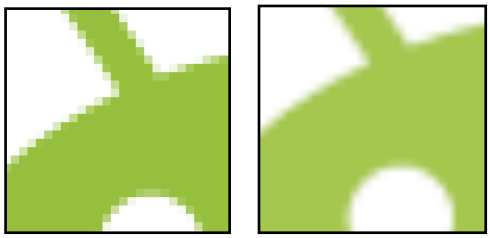
Las imágenes se almacenan en la carpeta *res/drawable-nodpi*, esta es una carpeta que indica que estas imágenes no dependen del tamaño de la pantalla del dispositivo que utilizamos y por lo tanto que siempre se usan las mismas. Esto es debido a que, como veremos más adelante, la API BTR ya se encarga de escalar la pantalla y con ello la imagen a las dimensiones admitidas por el dispositivo. A su vez, esto tiene sus pros y sus contras.

Por un lado, no es necesario tener varias imágenes iguales para cada escalado que necesitemos para que el producto funcione en las diferentes resoluciones admitidas por Android. Pero, si la resolución es mayor que el dibujo, este quedará pixelado, de forma que siempre tendremos que tener la versión con más resolución en memoria. Esto significa más uso de la RAM, puesto que, como hemos visto, las texturas se cargan directamente en la RAM de la gráfica, uso que en dispositivos de menos capacidad puede ser un problema. Aun así, el desarrollador siempre puede elegir tener las texturas de diferentes tamaños en esa misma carpeta y elegir él cual cargar según el dispositivo que se esté utilizando.

Veamos ahora como se cargan las texturas. Se utiliza la librería de Android *Bitmap Factory* [1], para descomponer la imagen en un mapa de bits en su escala original y que OpenGL pueda operar con él. A este mapa de bits se le asigna un nombre y se carga en la memoria reservada a OpenGL, que es como ya se ha dicho en la RAM de la gráfica del dispositivo. Luego se obliga a Dalvik a liberar la memoria que almacenaba el mapa de bits, de forma que no lo postergue a algún momento durante la ejecución del juego.

Volvemos al tema de los tamaños y las pantallas. Como ya sabemos la imagen no tiene escalado de ningún tipo, será cuando OpenGL la ajuste al tamaño de pantalla, cuando esta se reduzca o se amplíe. Para ello OpenGL necesita saber como actuar en que caso. Lo ideal sería que al reducirse no perdieran detalle y cuando se ampliase conservasen la calidad. Por eso se ha decidido en este trabajo utilizar mipmaps para los escalados inferiores y filtros bilineales para los aumentos.

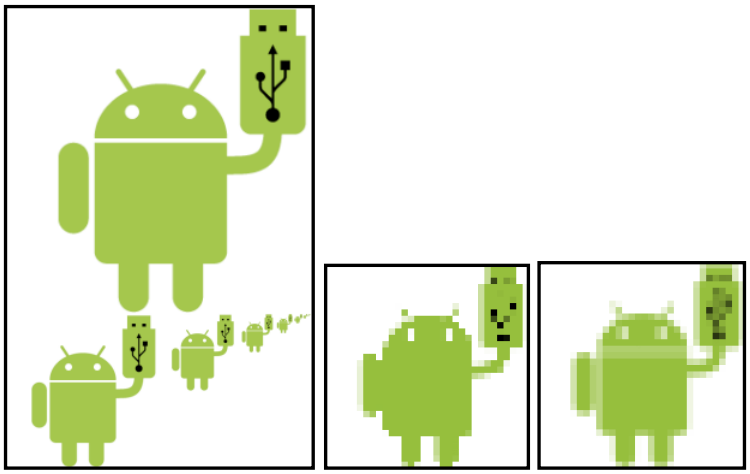
Hablemos primero del aumento, los filtros biliniales utilizan los 4 texel, la unidad más pequeña de medida de una textura sobre una superficie, más cercanos interpolándolos para obtener el textel completo. Esto funciona muy bien cuando tenemos objetos que se mueven en pantalla ya que si utilizamos el filtro de cercanía, es cierto que la imagen sería más nítida, pero puede que al moverla la situemos en una zona en la que un pixel tuviera que representar 2 textels, esto no es posible y la imagen sufre pequeñas deformaciones dando una sensación de mal funcionamiento.



Figuras 15 y 16: La primera representa el filtro de cercanía y la segunda el bilineal.

El filtro bilinial, funciona bien para magnificar, pero cuando se utiliza para reducir, es contraproducente porque cada vez más texels se acumulan en un mismo pixel resultando en una perdida total de la forma del la imagen, es por eso que en este caso se han utilizado los mipmaps. Los mipmaps lo que hacen, es crear un set de texturas de diferentes tamaños, para que a la hora de dibujar, OpenGL seleccione el que mejor se ajusta al espacio disponible para ser dibujado. Es cierto que los mipmaps ocupan más espacio en memoria, pero son ciertamente más rápidos en tiempo de ejecución, ya que no es necesario filtrarlos para ajustar, simplemente se coge el mejor en cada caso.

[1] http://developer.android.com/reference/android/graphics/BitmapFactory.html

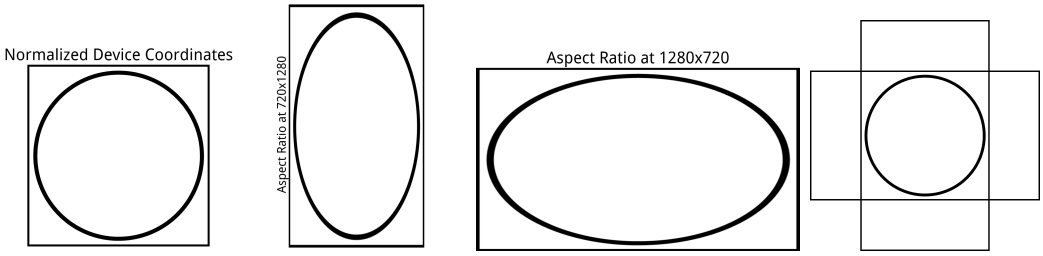


Figuras 17, 18 y 19: La primera representa los mipmaps, la segunda como quedaría una reducción con un filtro bilineal y la tercera como queda con un mipmap.

Todo el trabajo de texturas lo realiza la clase TextureHelper, encargada de leer la textura y cargarla en el buffer con la configuración especificada en este apartado. Posteriormente la clase devolverá la ID que OpenGL asigno a la textura para que se la pueda llamar desde los objetos que la usarán.

4.11. La pantalla y la proyección

Una vez se ha visto el proceso por el cual dibuja OpenGL es necesario hablar de la pantalla y de la proyección. El mayor problema que surge con OpenGL siendo multidispositivo, es la proporción con la que trabajamos. Como hay múltiples dispositivos y diferentes resoluciones, pantallas más panorámicas y otras más cuadriculadas, los objetos pueden quedar distorsionados dependiendo de nuestro tipo de pantalla y la relación de aspecto. Es por eso que no podemos trabajar constantemente con los mismos valores y es necesario normalizar las coordenadas para cada dispositivo.



Figuras 20, 21, 22 y 23: El circulo normalizado, lo que sucede si no se normaliza y que sucedería al girar el dispositivo en un entorno normalizado.

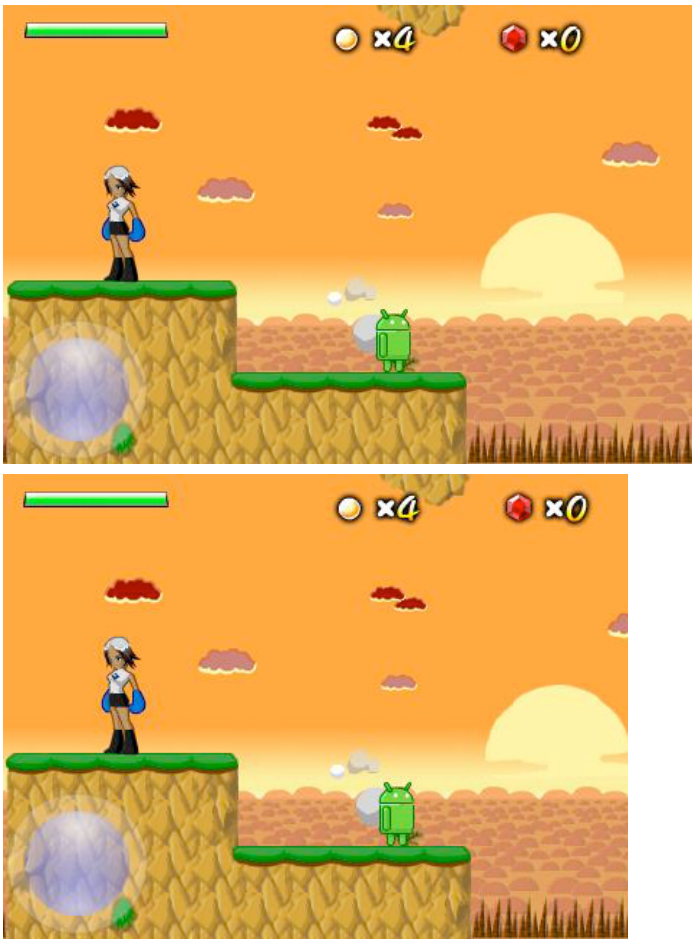
Por fortuna, para normalizar las coordenadas, solo es necesario saber el tamaño de la pantalla con la que estamos trabajando puesto que el propio OpenGL ya se encarga de la resolución. Sin embargo, todos los dispositivos tienen que tener algo en común, y eso eso el tamaño que va a representar nuestra pantalla OpenGL.

En este proyecto, se ha implementado que el desarrollador pueda elegir la dimensión de la parte más pequeña de su pantalla y de esta forma controlar el numero de metros representados por la pantalla y que así sea más fácil luego tener una escala para las medidas a usar. Saber que la

representación del ancho de la pantalla, por ejemplo, ocupa 10 metros, es muy útil sobre todo a la hora de utilizar físicas o escalar objetos. El tamaño por defecto esta entre las coordenadas -1/1.

Como se estaba explicando, para normalizar la pantalla simplemente se hace la regla de tres con el ancho y el largo del dispositivo. Por ejemplo, con una pantalla de 1280x720, si estamos en formato normal tendremos [-x,x] para el ancho y [-x*1280/720, x*1280/720], donde x es el numero de metros elegidos para representar. Y si pasamos a formato panorámico, tendríamos [-x,x] para la altura y [-x*720/1280, x*720/1280] para el ancho.

Siguiendo las formulas anteriores se pueden apreciar dos cosas. Por un lado, ciertos dispositivos tendrán acceso a una mayor porción del mundo virtual, esto es inevitable, y es una decisión que se toma en la mayoría de videojuegos móviles. Esto no tiene ningún impacto en el rendimiento puesto que OpenGL no procesa los objetos que están fuera de la pantalla. Por otro, que la verdadera dimensión de la pantalla será x*2 en su lado más pequeño.



Figuras 24 y 25: La porción del mundo disminuirá según el tamaño de la pantalla del dispositivo.

Una vez normalizada la pantalla se elige que tipo de representación queremos para los objetos que se hayan en el espacio generado. Para esto debemos hablar de la proyección.

Una proyección es una técnica de dibujo, para representar un objeto en una superficie, en nuestro caso los objetos se representan en la superficie de la pantalla. Como no vamos a trabajar en 3D sino solo en 2D y no hacemos uso de la lejanía, distancia que nos indica la coordenada Z, utilizamos una proyección ortográfica u ortogonal. En esta proyección, los objetos aparecen del

mismo tamaño sea cual sea su distancia con el origen o la cámara. Aquí es donde entran en juego la matrices y el álgebra lineal. Vamos a utilizar 3 matrices:

- **Matriz de proyección**: Es de la matriz que define las proyecciones, en este caso va a ser una proyección ortogonal, pero si vamos hacer 3D también podría ser una proyección en perspectiva, por ejemplo isométrica.

- **Matriz de modelo**: Esta matriz solo va a afectar al modelo, el modelo es el objeto, y nos permitirá hacer modificaciones sobre él sin que afecte al resto de la imagen. Modificaciones como rotaciones, translaciones, etc.

- **Matriz de vista**: Esta matriz representará la cámara y nos permitirá moverla por la escena.

4.12. El uso de matrices

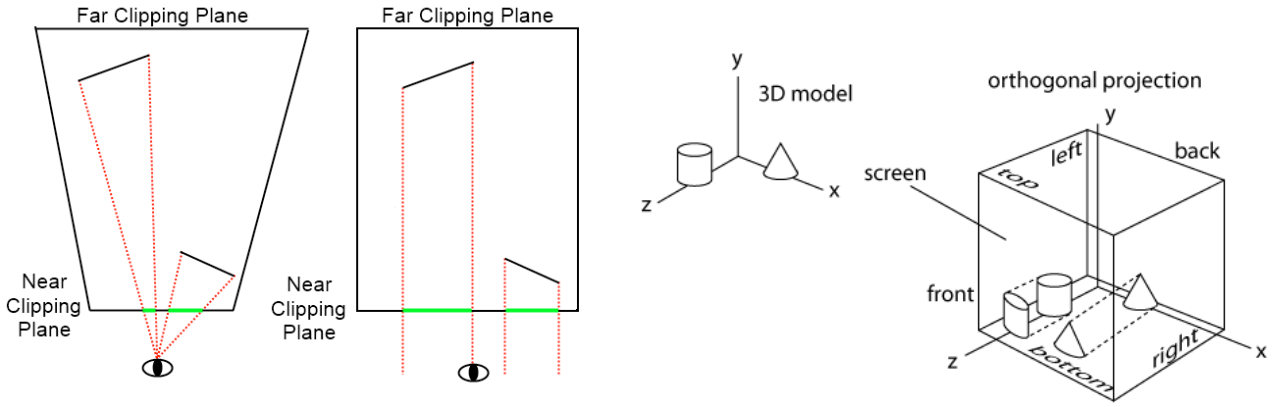
Ya se ha visto en el apartado anterior que OpenGL trabaja con matrices. Si nos acordamos de la sección de los shaders, recordaremos que el ejemplo de los valores uniformes era una matriz, esta matriz, es la resultante de operar todas las matrices mencionadas en el apartado anterior, y es la que afecta a los vértices de los objetos para situarlos en la posición correcta que deben tener en pantalla.

Las matrices tienen una dimensión de 4x4, y se multiplican por los vectores de 4 dimensiones que representan los vértices. Es cierto que OpenGL trabaja solo en 3 dimensiones, sin embargo sus vectores son de 4 (x,y,z,w). Esto es debido a que la w en realidad es un valor adicional para hacer las operaciones matemáticas más simples y poder crear perspectiva. Se denominan coordenadas homogéneas. En este caso no hay que tener en cuenta este valor, que será siempre 1.

La primera matriz a la cual se multiplican las demás es la matriz proyección. Como ya hemos dicho, la proyección va a ser ortonormal, u ortográfica. La siguiente matriz es representativa de este tipo de conversión. Donde r y l son derecha e izquierda, t y b arriba y abajo y f y n cerca y lejos:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Para hacernos una idea espacial de lo que hace esta proyección diremos que básicamente crea dos planos entre los que van a estar todos los objetos que represente nuestra pantalla. Luego hace un sandwich entre ellos sin tener en cuenta la distancia existente entre dichos objetos. La siguientes imágenes son representativas de ello y lo comparan con una perspectiva isométrica.



Figuras 26 y 27: Estas figuras dan una visión en 3D de la conversión en una proyección ortográfica.

En este proyecto esta matriz se configura en la clase GLRenderer, concretamente en el método visto al principio llamado *onSurfaceChanged*, y es que, es en ese método, en el que se especifica el tamaño de la pantalla con la que se va a trabajar, una vez se ha ajustado, no cambia hasta que no se rehaga la actividad o se produzca un cambio en la orientación del dispositivo.

La matriz de proyección se multiplica por la matriz de la vista. La vista representa una cámara, un ojo desde el que observamos la escena. Se especifica la posición desde donde miramos, la posición a donde miramos y donde está la cabeza de la cámara, es decir, si el vector UP mira hacia arriba la cámara estará de pie.

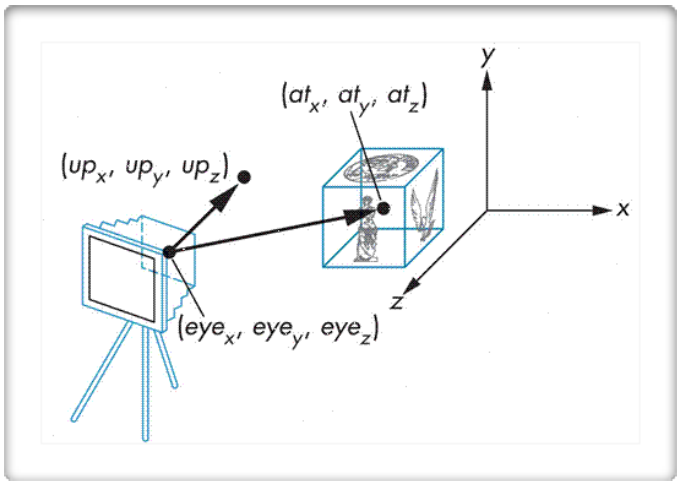


Figura 28: Representación espacial de la cámara.

Para este trabajo se ha especificado que la cámara este de frente al centro con la lejanía suficiente para que se vean los objetos, de otra forma al no haber espacio entre el centro y la cámara no se vería nada.

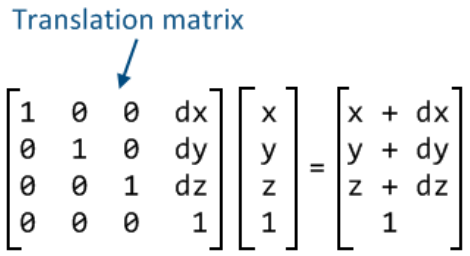
La cámara está implementada en una clase propia y única para ella misma. Esta clase actúa como si de una cámara de verdad se tratase y permite mover la cámara en el espacio como si fuera un objeto físico o hacer zoom. Posteriormente la clase GLRender se encarga de tenerla en cuenta a la hora de dibujar la escena.

Por último nos queda la matriz modelo que varía para cada objeto a dibujar y que aplica variaciones en dicho modelo. Es la ultima matriz por la que se multiplican las matrices anteriores y da lugar a la matriz que utilizará el programa de vértices, llamada matriz modelo-vista-proyección.

La matriz modelo parte de la matriz identidad, por si no se aplica ningún cambio, y sino, se la modifica con matrices de translación, rotación y escalado según las peticiones que haya hecho el desarrollador del nivel superior. Veamos todas las matrices utilizadas para construir la matriz modelo:

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- **La matriz identidad**: Los vectores multiplicados por ella no varían.



- **La matriz de Translación**: Los vectores multiplicados por ella se desplazan en el espacio. Afecta tanto a la x como a la y, y es la forma de mover los objetos por la pantalla

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & -\sin(a) & 0 \\ 0 & \sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **La matriz de rotación**: Se rota el vector por el eje de abscisas. Al ser una librería que trabaja sobre 2D las rotaciones solo se ejercen en el eje de las x.

$$\begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **La matriz de escalado**: El vector aumenta la dimensión especificada. En este caso Sx y Sy son iguales ya que el escalado se produce sobre x e y al mismo tiempo, de esta forma se evita que se deforme la imagen.

Todas estas funciones han sido implementadas en la clase ModeledObject. La clase almacena todas las variables que tiene a su disposición el desarollador y hace los cálculos necesarios a la hora de dibujar para devolver una matriz que cumpla los requisitos. Esta clase la extienden todas las clases que forman los objetos disponibles en la API BTR.

4.13. La animación

Ahora que se ha formado una idea básica del funcionamiento de OpenGL en este trabajo. Es necesario centrarnos en los sprites y empezar a hablar de animaciones y de las capas más altas de la librería implementada, que son con las que trabajarán futuros desarrolladores.

Figura 29: Ejemplo de secuencia animada.

Lo que vemos aquí arriba es una secuencia animada en 2D. Para que de una sensación de movimiento, lo que hay que hacer es cambiar la textura del objeto en un determinado intervalo de tiempo.

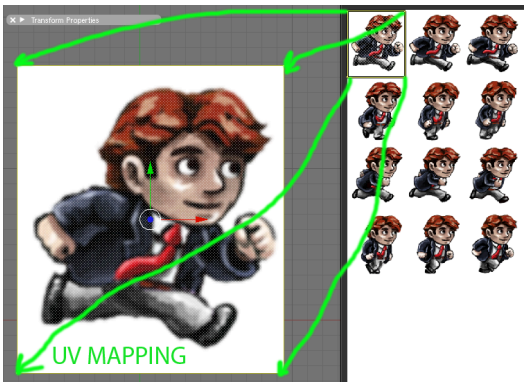


Figura 30: Ejemplo de coger una porción de textura para un sprite.

Sin embargo, cambiar la textura sería altamente ineficiente, y lo que se hace es introducir en la misma textura todos los frames necesarios para conseguir esa animación. De esta forma, cada vez que cambia el frame, OpenGL no necesita buscar una nueva textura y una nueva dirección de memoria, sino que simplemente nos muestra una porción de la textura utilizada. Aquí es donde vuelven a aparecer las shaders, concretamente las de vértices, pues es ahí, como ya se ha comentado, donde se especifican las coordenadas de las texturas que se van a utilizar dentro de las propias texturas.

Por cada frame que renderice OpenGL, se pasaran nuevos valores de las coordenadas en las cuales se encuentra la fracción de textura que se va a utilizar. Esto es muy sencillo de ver y solo tiene una pequeña anomalía, y es que, en las texturas, la coordenada Y aumenta hacia abajo, es decir, el origen de coordenadas esta en la punta superior izquierda y la imagen termina en la punta inferior derecha. Esto hay que tenerlo en cuenta cuando queramos especificar donde se encuentra un determinado sprite en un mapa de sprites. Aun así, de momento esto no nos preocupas puesto que la API BTR esta programada de ese modo y solo requiere que se especifique el tamaño del sprite, el numero total de frames y el numero de filas y de columnas.

Todo esto se implementa en la clase AnimatedSpriteObject, de la cual se hablará a continuación.

4.14. Los diferentes tipos de sprites

Hablemos ahora de los diferentes tipos de objetos implementados:

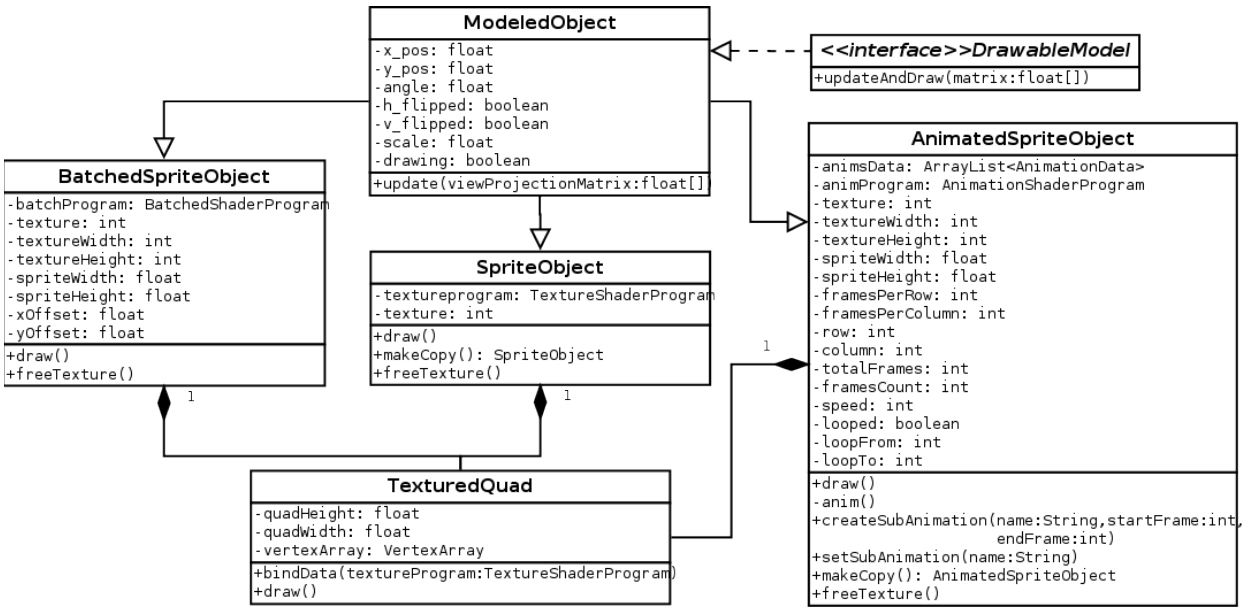


Figura 31: Diagrama de clases que implementan los objetos.

Como se ve en el diagrama todos ellos parten de la clase Modeled Object, que es la clase que se menciono antes que preparaba la matriz modelo para poder actuar sobre el objeto exclusivamente.

- **Simple**: Este es el caso más básico, simplemente sitúa una textura sobre una superficie rectangular. Se utiliza sobre todo para objetos estáticos, como fondos, interfaz, etc.

- **Animated**: Este se utiliza para sprites animados. Utiliza una única textura donde están todos los frames de los que dispone la animación y permite moverse a través de ellos de formas diversas. De esta forma pueden existir varias animaciones dentro de un mismo objeto lo que permitirían por ejemplo que un personaje ande, salte y de golpes, cambiando solo una variable. Este tipo de sprite tiene cierta restricciones de tamaño. Para hacer más fácil y sencillo al programador poder incluir animaciones, solo se mueve entre los frames existentes, limitando el tamaño a uno único para cada sprite creado de este modo.

- **Batched**: Este sprite se acerca al animado en el sentido de que la textura se saca a partir de un mapa más grande de texturas, de ahí el vocablo batched, del ingles agrupación. Pero en este caso el programador tiene más libertad para moverse por él. El programador ahora trabaja con las coordenadas del mapa de texturas. Esta forma de trabajar es muy eficiente, una vez la textura esta cargada en la RAM, la dirección de memoria está almacenada y no tiene que hacer búsquedas, es por eso que se ha implementado este tipo de sprite, para la eficiencia.

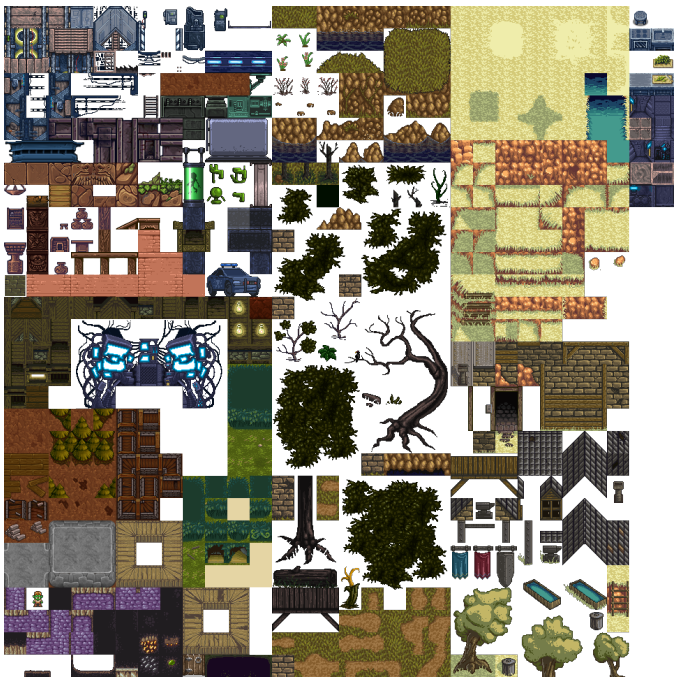


Figura 32: Ejemplo de agrupación de sprites.

Estas clase terminan de enmascarar el proceso de creación de un objeto en pantalla. En ellas se crea el programa, se prepara el rectángulo y la textura que se inserta en él. A la hora de dibujar efectúan el trabajo necesario para llevar a cabo el procedimiento y que el objeto aparezca por pantalla sin que el desarrollador de alto nivel deba preocuparse de nada.

5. La API: Blue Tree Reality

Todo lo mencionado anteriormente hace referencia a OpenGL y a su modo de trabajar. Sin embargo, la API BTR no espera que el desarrollador de alto nivel tenga que lidiar con ningún aspecto propio de OpenGL y oculta su trabajo interno, tal y como vimos en las secciones anteriores. Aun así, queda trabajo por hacer para manejar todo lo visto. Esta capa superior, la API BTR, hace que ese proceso sea sencillo.

El desarrollador podrá controlar los sprites vistos y la pantallas que contenga el juego, gracias a las siguientes clases de la librería creada.

5.1. Sprite Manager

El Sprite manager permite la creación, selección y destrucción de sprites. Sprites, que a su vez se dividen en dos clases. Los sprites de tipo normal, y los sprites que pertenecen a la HUD, es decir, la información que en todo momento se muestra en pantalla durante la partida, como por ejemplo, la vida del jugador o los controles. La principal diferencia es que a los sprites que están en la HUD no les afecta el movimiento de la cámara, ni siquiera el zoom y además están siempre al frente, por encima de los sprites normales.

Figura 33: Ejemplo de HUD. En ella se analiza la diferente información que tiene un jugador durante el juego.

Los sprites se pintan en el orden en que se introducen, es decir, los fondos sería lo primero en introducirse y las figuras que están al frente lo último tapando todo lo que se haya introducido con anterioridad. El gestor de sprites se encarga de pintarlos en ese orden a través de la clase GLRenderer de OpenGL.

Los métodos del gestor permiten acceder a los sprites para cambiar sus propiedades. Esto se realiza a través de los controladores de los sprites.

5.2. Sprite Controller

Esta clase se encarga de hacer de intermediario entre el Sprite y todas sus funciones. En este trabajo se a optado por darle una ID o nombre, en forma de String a cada sprite, para poder

referenciar los objetos de forma sencilla. Así, es posible definir constantes con los nombres de cada objeto y acceder a ellos en cualquier momento utilizando su referencia.

Haciendo una petición al *Sprite Manager* con una determinada ID o Nombre, se obtiene el controlador al cual pertenece esa ID. Esto produce un problema, en el caso de haber 2 controladores con la misma ID devolvería el primero que encuentre. Es por eso que el desarrollador de la capa superior no debe utilizar Sprites con la misma ID ya que le sería imposible acceder a uno de ellos.

5.3. Sprite Resources

Toda la información que se almacena y que maneja el gestor de sprites está en esta clase. Esta es la clase que reserva la memoria y que se encarga de facilitar al gestor las diversas opciones de las que hace uso. Esta clase debe implementarse para cada pantalla y la rellena el desarrollador de alto nivel con los recursos que desee utilizar, en concreto, deberá completar los siguientes métodos:

- **resourcesToLoad**: Lugar donde se cargan los Sprites a utilizar en esa pantalla. Deberá especificar todos los recursos a usar. Incluso las texturas de aquellos objetos que se vayan a crear durante la ejecución. Simplemente tiene que nombrarlos y añadirlos tal y como se indica en el manual[ANEXO 1].

- **resourcesToFree**: Lugar donde se liberan las texturas cargadas. De nuevo esto se menciona en el manual. Las texturas solo es necesario liberarlas 1 sola vez, es por eso que las copias y los objetos que comparten textura no hace falta que sean liberados múltiples veces.

El resto de la memoria utilizada la libera el gestor cuando se cambia de pantalla. La memoria de OpenGL no se libera, simplemente se marca como reutilizable y a la hora de hacer nuevas inserciones se sobrescribe.

Nota: Es importante recalcar la creación del método makeCopy, para un sprite ya creado. Cuando queremos un sprite igual que otro utilizamos este método, de esta forma se comparten muchas de los recursos y no se introduce 2 veces la textura en la memoria, esto sería totalmente ineficiente e inútil, puesto que solo conseguiríamos a OpenGL a cambiar sus referencias a la hora de pintar dos objetos iguales.

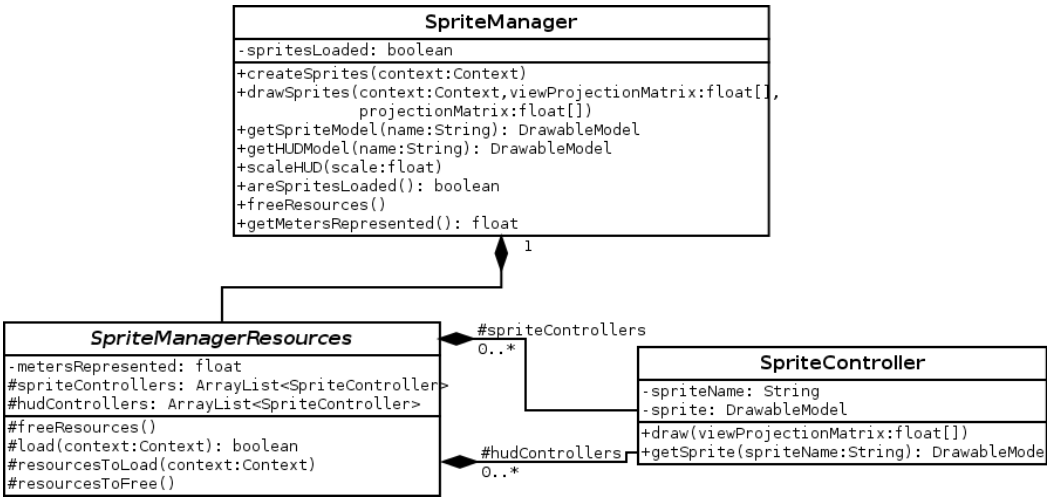


Figura 34: Diagrama de clases representativo del gestor de sprites

5.4. Screen Manager

Igual que hay un gestor de Sprites también hay un gestor de pantallas.

Puesto que ya hemos creado una actividad con una pantalla de OpenGL no es necesario crear una nueva actividad cada vez que se quiere cambiar de pantalla. La API BTR ha sido diseñada para que esto se realice desde dentro y no sea necesario volver a cargar OpenGL por cada pantalla diferente, lo cual sería lento. Es por eso que la API BTR funciona con clases Screen, del ingles pantalla y con un Screen manager, o gestor de pantallas.

Como se ha visto en el apartado anterior, el gestor de Sprites libera y carga los recursos directamente cada vez que tenga que usar una pantalla o deje de utilizarla, de este modo no sobrecargaremos la memoria disponible y será como si cada pantalla fuera una pantalla nueva. Simplemente hay que indicarle que la screen ha terminado, e indicarle cual es la siguiente screen a utilizar.

En su diseño, el gestor de pantallas se asemeja mucho al gestor de Sprites, las pantallas vuelven a ser elementos individuales manejadas por un controlador, al cual se accede con una ID o nombre.

Las pantallas se controlan con métodos propios a estas que implementa la interfaz *ScreenInterface*. El desarrollador, deberá especificar cual es la siguiente pantalla a la que se accederá indicando su nombre, y marcar la pantalla actual como terminada. Una vez echo esto, la API BTR se encargará del resto.

Todas las pantallas constan de los siguientes métodos a completar por el desarrollador:

- **updateLogic**: Este método y el método que se explicará a continuación son métodos a los que el bucle de juego llama continuamente. En estos métodos es donde se producen todos los cambios del juego 60 veces por segundo. En este caso, solo se debe actualizar la lógica del juego, es decir la parte que no es visual. Se deben resolver todas las variables que afectan al juego, como por ejemplo: el movimiento, la posición, las colisiones, las físicas, etc. Una vez se obtienen todos estos datos ya se puede dibujar.

- **updateSprites**: En este método se actualizan, a raíz de todos los cambios producidos en el método anterior, todo lo que sea visual. Por ejemplo, si se ha desplazado un sprite varios metros en la pantalla, se deberá utilizar el método para especificar la nueva posición de ese sprite en pantalla. Esto no significa que en este método se dibuje nada por pantalla, simplemente se actualiza la información de los sprites para cuando se dibuje.

- **onOnTouchEvent**: Desde este método se manejan los eventos táctiles que se reciben por pantalla. Es muy importante saber que las coordenadas que llegan por pantalla no son las mismas que las que utiliza el OpenGL. Sin embargo, esto no es problema y la conversión es sencilla. El desarrollador puede desde ese mismo método obtener las dimensiones de la pantalla y hacer la conversión con las dimensiones que él mismo especifico en la sección de recursos.

- **onOnDrawCreation**: Al ser imposible manejar OpenGL desde fuera de su hilo y Android no permite almacenar la dirección de memoria de las variables que se usan en este, la única forma de

crear nuevos sprites una vez se ha iniciado la pantalla es durante la fase de dibujo. Es por eso que se implemento este método, ya que en juegos en los que se repiten muchos sprites y no se tiene uno numero estimado de ellos para reservar desde el principio, es necesario crear durante la ejecución de la pantalla. Además este método también se puede usar para crear pantallas de carga en las se cargen sprites de otras pantallas, permitiendo una representación visual de dichos tiempos de espera.

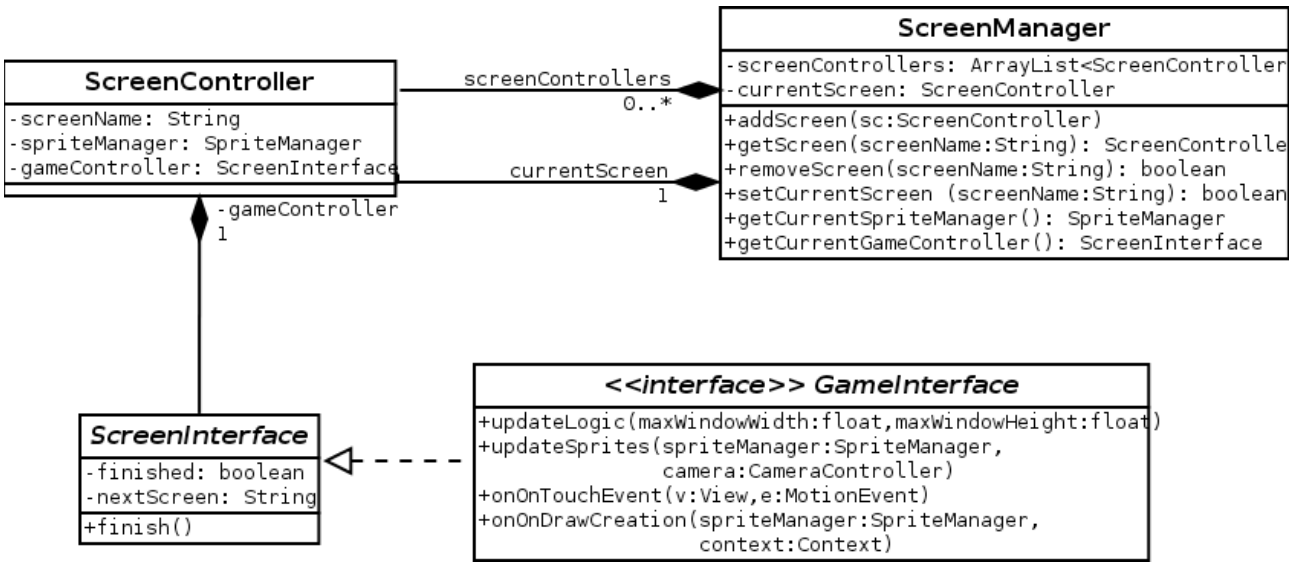


Figura 35: Diagrama de clases representativo del sistema de pantallas.

5.5. El bucle de juego y los hilos

Una vez tenemos todos los elementos que forman el juego, necesitamos ponerlos en funcionamiento. Para ello necesitamos el bucle del juego, que es el corazón que marca el tiempo en el que este se desarrollan las acciones. Todas las clases anteriormente mencionadas están presentes en este bucle, trabajando a las ordenes de la API BTR.

Durante el tiempo de juego hay 3 hilos funcionando:

- El **hilo principal de Android** que controla todos los procesos subyacentes y de la actividad y del cual no se hace uso directo.

- El **hilo del bucle del juego**, que se crea al principio y que controla todos los cambios que se producen en la lógica del juego y que afecten a la forma en que se dibuja.

- Y el **hilo que crea OpenGL**, en el cual se refresca la pantalla con las actualizaciones producidas durante el juego.

En principio el bucle del juego se actualiza 60 veces por minuto, lo que no equivale a 60 FPS, sino a 60 UPS. Esto quiere decir que de lo único que esta seguro el desarrollado es que si las actualizaciones del juego no requiere muchos recursos y se ejecuta a una velocidad aceptable, se actualizará 60 veces por segundo. En ese mismo bucle se solicita dibujar cada vez que termina una actualización, pero es el sistema quien decide cuando va a dibujar.

El tiempo anteriormente mencionado se controla tomando el tiempo de inicio, el de final y en caso de haberlo hecho en menos de 1/60 segundos, se pausa el tiempo sobrante. Se ha elegido 60 FPS por ser un estándar en la industria.

Que sea Android el que elija cuando se dibuja tiene sus pros y sus contras. Es una ventaja, en el sentido de que el sistema ya calcula la tasa de refresco de la pantalla y controla los recursos para que el movimiento vaya fluido decidiendo, o no, saltarse frames según convenga(esto es imperceptible al ojo humano, que aproximadamente ve 30 FPS). El punto malo, es que a la hora de tener un verdadero problema de actualizaciones, no se puede tener el control absoluto para solventarlo. Normalmente, lo que se hace es saltarse frames para terminar las actualizaciones de la lógica y volver a dibujar una vez se haya equilibrado. En este caso, se han hecho pruebas, las cuales se comentarán más adelante, con esta metodología, pero la diferencia no es realmente notable. Aun así, la optimización previa en la construcción de las clases de sprites permite tener una cantidad notable de objetos en pantalla con diferentes texturas sin que decaiga el rendimiento del sistema.

Otro factor de riesgo con el tema de los hilos, es el traslado de información de un hilo a otro. Se está utilizando el método *syncronize* con la clase *Sprite Manager*. Sobre esta clase se producen todas las actualizaciones que tiene que ver con el dibujo, luego cuando llega a la función en la que pinta la pantalla, se accede a esta clase bloqueando las actualizaciones provenientes del bucle del juego.

De nuevo volvemos al rendimiento, como no se sabe en que momento se dibuja está clase podría dibujarse mientras se están actualizando ciertos objetos, es por eso que debe ser bloqueante o sino podríamos tener desplazamientos de objetos incorrectos. Recordemos que lo más importante es el update y esto permite que se actualice y se dibuje a la vez sin que se estorben.

Ahora surge la pregunta, ¿que pasaría si tardase mucho en dibujar? Es por esto que se hace útil controlar que si el dibujo tardase mucho en realizarse, el update, no se debiera seguir enviando peticiones para dibujar y siguiera adelante él solo hasta estabilizar la velocidad. Sin embargo, esta medida no es necesaria ya que como se dijo anteriormente es el sistema el que controla como va la velocidad del sistema y le da prioridad o no al dibujo. Por eso, si tarda demasiado en dibujar, él mismo le impide hacerlo hasta que el resto del sistema se estabilice. Aun así, en caso de que el dibujo se hiciera muy pesado es inevitable que sintamos retrasos durante el juego, a no ser que sea un evento pasajero y no realmente una sobrecarga en el dibujo.

6. Las colisiones

Las colisiones son representaciones matemáticas del espacio que ocupa un cuerpo. Se utilizan para detectar si se están superponiendo con otra colisión y de esta forma saber si se ha producido un "choque" entre dos objetos.

Hay muchas formas de calcular colisiones. Cuando los juegos utilizaban cuadrículas o mallas para producir las escenas, se hacían los cálculos utilizando esas mismas formas. Hay juegos como el Worms que incluso implementaron un sistema de colisión por pixeles. Hoy en día, se utilizan las formas envolventes, y es lo que se a utilizado en este trabajo.

Se han implementado las 2 formas más representativas y eficientes, los rectángulos y los círculos. En caso de que estas formas no se ajusten bien a un objeto del que se precise más nivel de detalle, siempre se pueden usar varias de estas formas para crear más realismo.

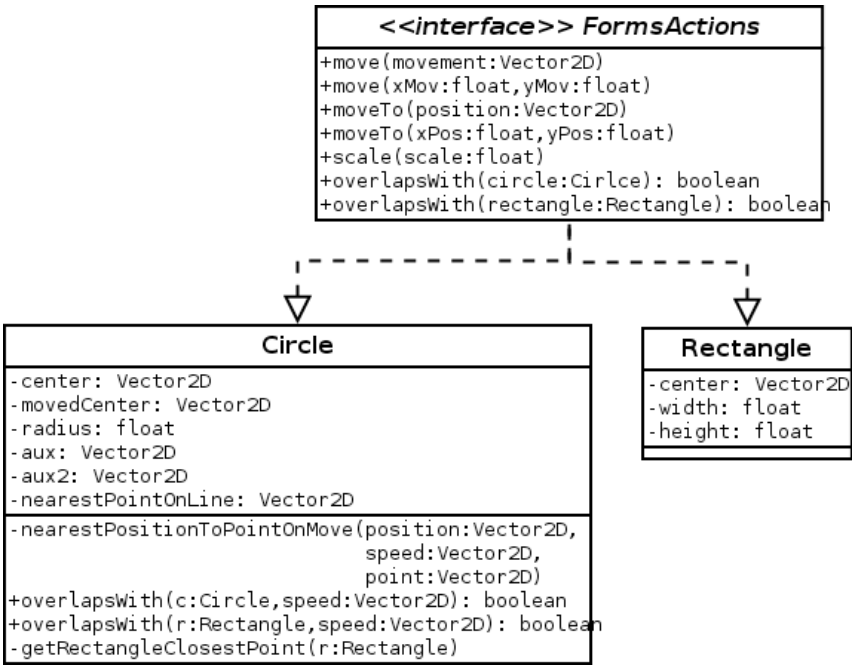


Figura 36: Diagrama de clases de las colisiones.

6.1. Vectores

Tanto para esta sección como para la de físicas se ha implementado una clase representativa de vectores en dos dimensiones. Esta clase se denomina *Vector2D* y permite controlar los vectores con las siguientes operaciones: Suma entre vectores, Resta entre vectores, multiplicación por un escalar, producto escalar, longitud de dicho vector, normalización del vector, ángulo con respecto al origen de coordenadas, rotación del vector un determinado ángulo, distancia entre 2 vectores y la distancia entre 2 vectores al cuadrado.

Esta ultima operación se implemento por temas de eficiencia, realizar la raíz cuadrada de un número con la librería matemática, requiere un gasto de procesamiento innecesario en ciertos caso de comparación. Por lo tanto, se omite y solo se recibe la distancia al cuadrado.

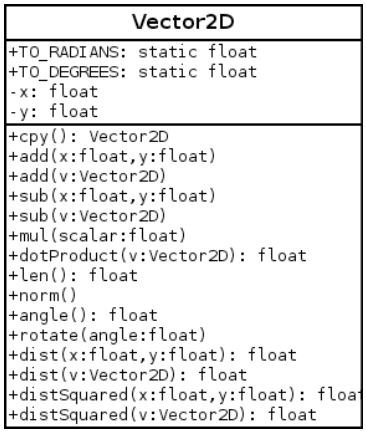


Figura 37: Clase Vector2D.

6.2. Rectángulos

Los rectángulos se forman especificando su altura y anchura. Luego se colocan en el espacio especificando su centro.

Figura 38: Representación de una colisión rectangular, marcada en rojo, sobre un cuerpo.

Para comprobar si dos rectángulos colisionan se calculan sus vértices superior e inferior y se comprueba que el vértice superior derecho del primer rectángulo sea más grande que el inferior izquierdo del otro y que su vértice inferior izquierdo sea más pequeño que el vértice superior derecho del otro. Esta forma de cálculo asegura las colisiones incluso cuando un rectángulo contiene a otro.

Para comprobar si un rectángulo colisiona con un círculo, se obtienen las coordenadas más cercanas del rectángulo al centro del círculo, esto se realiza mediante comprobaciones. Si dichas coordenadas resultan estar dentro del área que del círculo, definida por su centro y radio, es que se ha producido una colisión.

6.3. Círculos

Los círculos se definen por su radio y al igual que los rectángulos se sitúan utilizando un centro. El método que se utiliza para calcular la colisión con otro rectángulo es idéntico al visto en el apartado anterior.

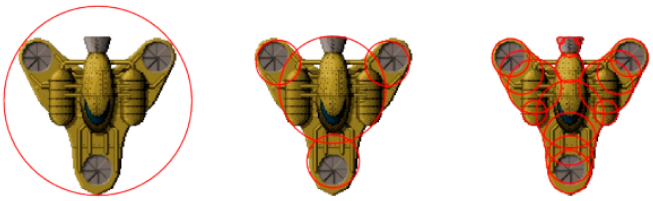
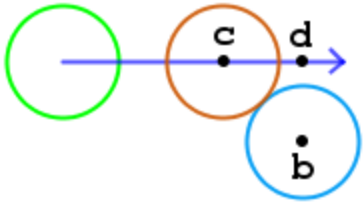
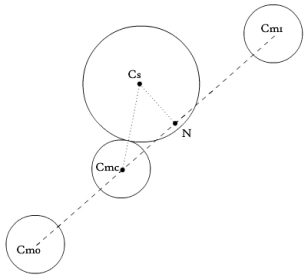


Figura 39: Representación de una colisión circular sobre un cuerpo, marcada en rojo, y sus posibles combinaciones.

La colisión con otro círculo se calcula comparando la distancia entre los centros con la suma de sus radios, si es inferior entonces colisionan. Este es un ejemplo de comprobación en la que se usa la distancia al cuadrado.

A los círculos se les ha añadido también colisión en movimiento. La colisión en movimiento contra otro círculo, se calcula obteniendo el punto, en la linea que describe la trayectoria, más cercano al centro del otro círculo. Se calcula si con el desplazamiento actual se llegaría en ese frame a dicho punto. Luego, se utiliza el Teorema de Pitágoras para obtener la distancia de ese punto al centro del círculo, si la distancia es inferior a los radios entonces colisionan. Afortunadamente, al utilizar este sistema, se puede calcular el punto en el que colisionan, de esta forma los círculos no se solapan.





Figuras 40 y 41: Representaciones visuales del calculo en movimiento. Mostrando el triángulo usado para calcular la distancia.

Cuando es un rectángulo la idea es la misma. Se obtiene el punto más cercano al rectángulo en la linea del movimiento, luego se comprueban todos los vértices de ese rectángulo para saber cual es el más cercano al centro del círculo. Desde ese vértice se trabaja para saber si el círculo ha colisionado con el rectángulo y en ese caso cuanto ha penetrado en él para retrocederle hasta el punto de la colisión, dejando el círculo en ese punto.

Con las funciones que calculan las colisiones con objetos en movimiento se evita el tunelling. El tunelling, del ingles tunel, consiste en objetos, que debido a su velocidad, cuando se hace la comprobación de colisión esta resulta haber pasado el objeto y no colisiona con él, creando un fallo. Este sistema, al calcular el punto más cercano de la trayectoria, evita este problema.

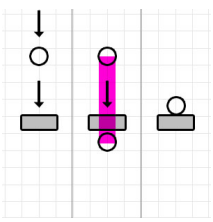


Figura 42: Representación del efecto tunelling.

7. Las Físicas

En este apartado vamos a ver la última implementación de ayudas al programador del juego. En este caso son las físicas, que hacen que los objetos interaccionen con el mundo de forma realista. Para ello, se ha hecho un a implementación de las físicas de Newton utilizando el siguiente modelo.

7.1. Verlet

La integración de Verlet es un método numérico usado para integrar las ecuaciones del movimiento de Newton. Esta es su fórmula:

Original Verlet:

$$x_{i+1} = x_i + (x_i - x_{i-1}) + a * dt * dt$$

Donde *x* es la posición, *a* la aceleración y la velocidad se calcula con respecto a las posiciones actual y posterior. Todo esto se implementa en la clase Verlet, que nos permite controlar de forma sencilla estos aspectos.

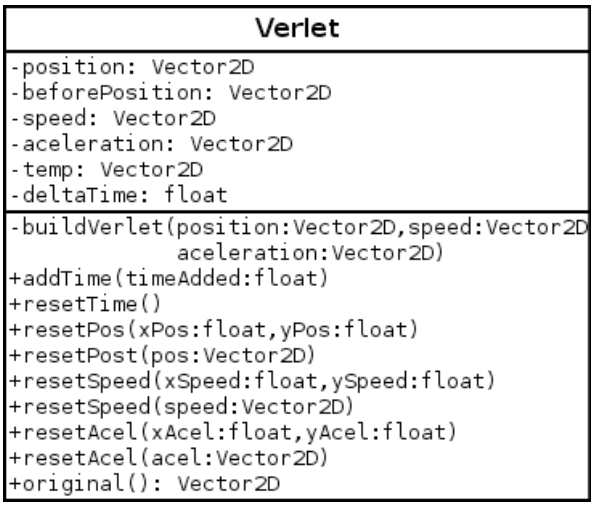


Figura 43: Clase Verlet.

Como se puede apreciar, se pueden hacer cambios de cualquier variable en todo momento. Esto se ha implementado para tener en cuenta las aceleraciones instantáneas que se pueden producir, por ejemplo, al recibir un golpe. De otra forma, el objeto describirá una trayectoria equilibrada.

Esta clase es la base del controlador de físicas del que hablaremos luego, ahora expliquemos la fuerzas implementadas.

7.2. Fuerzas

Se han implementado 3 fuerzas cada una como una clase: la fuerza normal, la fuerza de la gravedad y la fuerza de fricción. Con estas 3 fuerzas se consiguen todos los efectos.

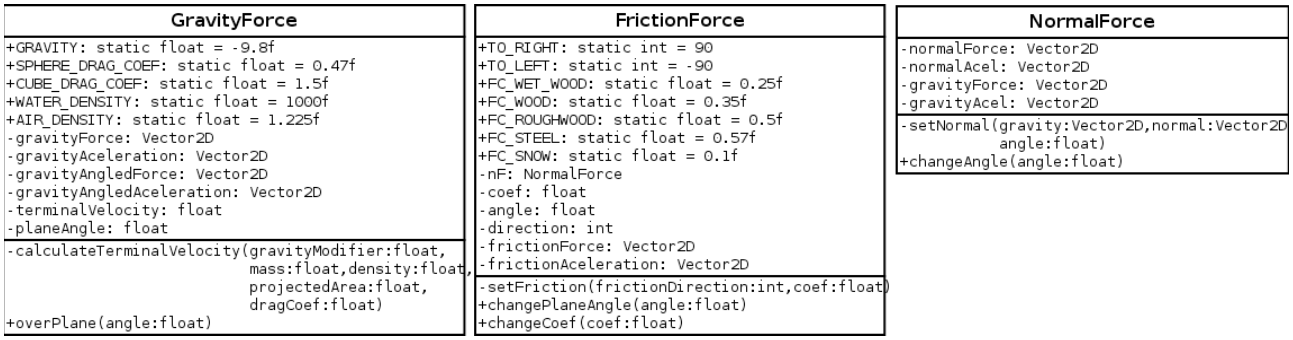


Figura 44: Clases que representan las fuerzas.

- **La fuerza de la gravedad o fuerza peso**: Es la fuerza que hace que los objetos caigan con una aceleración constante. Se calcula usando la siguiente ecuación:

$$\mathbf{F} = m \cdot \mathbf{g}$$

Donde *m* es la masa y *g* la constante de la gravedad.

Esto causaría que los objetos acelerasen hasta el infinito, por eso se ha implementado junto a dicha fuerza la velocidad terminal. La velocidad terminal es la velocidad máxima a la que podría llegar un cuerpo, dadas sus propiedades físicas, en una atmósfera determinada. La velocidad terminal se calcula utilizando la siguiente ecuación:

$$v_\infty = \sqrt{\frac{2F}{\rho A C_d}}$$

Donde, *F* es la fuerza peso, *p* es la densidad del fluido por el que se mueve el objeto, *A* es su área y *Cd* el coeficiente de resistencia aerodinámica.

La clase está diseñada para poder ampliarse y dar la posibilidad de cambiar todas estas variables. Por el momento, al crearse se utilizan los valores estándar para esferas en la atmósfera terrestre.

Esta clase devuelve tanto la fuerza de dicha gravedad como las aceleraciones provocadas por ella para poder hacer los cálculos precisos.

- **La fuerza Normal**: Es la fuerza que ejerce una superficie sobre el cuerpo apoyado sobre la misma. Responde a la siguiente ecuación:

$$F_n = mg\cos\alpha = \mathbf{P} \cdot \mathbf{n}$$

Donde, *m* es la masa del cuerpo, *g* la constante de la gravedad y *alfa* el ángulo de inclinación del plano sobre el que se encuentra.

En nuestro caso esta fuerza se utiliza para las caídas por planos inclinados.

- **La fuerza de fricción o rozamiento**: Es la fuerza que aparece cuando dos superficies entran en contacto y que se opone al desplazamiento de una de ellas. Se calcula utilizando la siguiente ecuación:

$$F_r = \mu N$$

Donde μ es la constante de fricción que varia dependiendo de la superficie, y N es la fuerza normal.

Esta clase es muy útil para el desplazamiento de objetos por planos inclinados, o la deceleración o aceleración progresiva por superficies horizontales debido a fuerzas instantáneas, como los golpes.

7.3. El controlador de físicas

Todas las clases anteriores se agrupan y controlan desde la clase denominada *ObjectPhysicsController*, del ingles controlador de físicas del objeto, lo cual especifica precisamente lo que hace. Todo objeto que responda a interacciones físicas con el entorno, deberá instanciar esta clase.

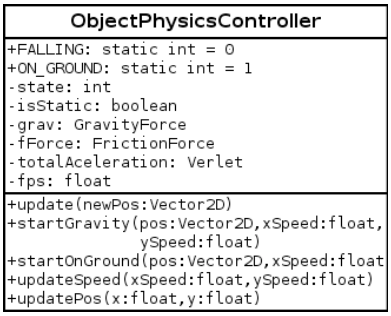


Figura 45: Clase de control de físicas.

Una vez instanciada, la clase permite ejercer gravedad en caída libre y rozamiento sobre un plano de forma sencilla. El programador solo deberá especificar que tipo de efecto desea y llamarla con cada actualización del juego para obtener la nueva posición. En caso de existir cambios por choques u otros motivos, siempre puede actualizar la velocidad y la posición.

8. Pruebas y Resultados

Para realizar todas las pruebas de este trabajo se a utilizado el Log de Android[1]. La mayoría de Logs se han ido retirando una vez se han completado las pruebas de dicha fase, pero aun así sigue existiendo, en el paquete *com.bluetreereality.openglapi.util* la clase *LogConfig* que maneja la activación de los logs de shaders, fps y texturas. Los cuales se muestran útiles para los futuros desarrolladores, sobre todo el de FPS.

Se han creado 5 entornos, los cuales sirven como ejemplos para futuros desarrolladores. En ellos no se produce nada específico y en algún caso se puede llegar a pensar que tienen un mal funcionamiento, sin embargo simplemente son campos de pruebas para realizar los cambios y ver su resultado, estos son:

Figura 46: Entorno de pruebas Braid Test.

- **braidtest**: Quizás sea el más cerrado de todos. En el se cargan unos sprites y se hacen modificaciones sobre ellos al tocar la pantalla. Este entorno es el único que hace uso de la cámara, moviéndola y utilizando el zoom a la vez que escala los elementos de la HUD.

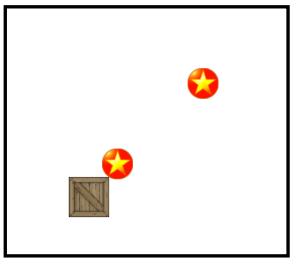


Figura 47: Entorno de pruebas de colisiones.

- **collisiontest**: Este es el entorno de pruebas para colisiones únicamente. En el hay objetos circulares y cuadrados que se amoldan a la perfección a los contornos representados por las colisiones y se puede ver de este modo como interactuan entre ellas.

[1] http://developer.android.com/reference/android/util/Log.html

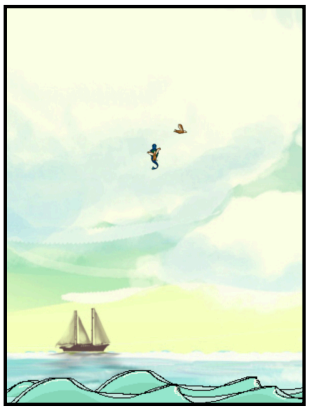


Figura 48: Entorno de pruebas Fish Jump.

- **fishjump**: Este es un pequeño juego creado en el cual, un pez salta para alcanzar un pájaro. Esta a caballo entre los dos anteriores pero también incorpora físicas.

Figura 49 y 50: Entorno de pruebas de menú y de carga respectivamente.

- **screenmanagement**: Este paquete incluye un menú de selección y una pantalla de carga. Es un buen ejemplo de una posible implementación de estos dos elementos muy comunes en videojuegos.

Figura 51: Entorno de pruebas de físicas.

- **spacetest**: En este entorno se realizaron todas las pruebas de físicas y algunas pruebas de rendimiento aumentando el numero de objetos y texturas para ver los límites de la librería creada.

Las pruebas de rendimiento de los sprites, se realizaron añadiendo objetos hasta que caían los FPSs. Estos objetos eran objetos animados que además utilizaban físicas. De esta forma, se comprobaba tanto el rendimiento al dibujar como el rendimiento al actualizar los procesos del bucle de juego. Esta tabla muestra los resultados:

Nº de Objetos	UPS	FPS
50	60	45
60	58	36
70	58	29
80	55	9

Tabla 2: Velocidades de procesamiento con respecto al numero de objetos activos en pantalla.

Estos resultados son los mismos tanto si se usan texturas diferentes como si se utilizan objetos copiados. La única diferencia en ese caso, es que exista o no una parada intermedia al generar cada objeto durante el tiempo de juego en el caso de no ser una copia.

Este bajón en el rendimiento fue notable a simple vista. Es necesario resaltar que estas pruebas fueron realizadas con el emulador, cuyo rendimiento es mucho más bajo que el de los móviles. Sin embargo, se considera que para un videojuego 2D y teniendo en cuenta que los UPS no decaen prácticamente se consideran un rendimiento eficiente, ya que como se ha mencionado anteriormente el proceso del dibujo lo controlan entre OpenGL y Android, la Api se limita a interactuar con ellos.

Para realizar las pruebas con el bucle del juego se introdujeron paradas de una cantidad notable de milisegundos que impidiese llegar a los 60 frames por segundo. Estos bloqueos se situaron tanto en el update como en el draw, indistintamente según la prueba. Efectivamente se notaba un desfase en el juego.

Tras un estudio detallado utilizando el Log e intentando aminorar la cantidad de desfase producida. Se concluyó que el control que ejercía Android sobre la velocidad con la que pintaba la pantalla era suficiente para problemas de este calibre, en el caso de que fuera el dibujo el que tardase en procesar. En el caso de que sea el propio update del juego, por intensidad de calculo, el que tarde en finalizar sin cumplir con sus tiempos el programador tiene un problema grave y a no ser que sea un proceso pasajero deberá enmendarlo.

9. Conclusiones

Efectivamente la librería gráfica OpenGL, necesaria para hacer videojuegos para dispositivos móviles, no es sencilla de utilizar. Este proyecto ha conseguido hacer uso de ella eficiente para videojuegos 2D, pero solo se ha rasgado la superficie de su potencial y es que, adentrarse en los mundos tridimensionales aumenta las dificultades exponencialmente. Además, la librería ha conseguido enmascarar por completo este aspecto del desarrollo de videojuegos 2D para facilitar una API que permita continuar con el resto de trabajo sin preocuparse.

Tanto las físicas como las colisiones también han sido un logro. Es necesario reforzar la idea de que para terminar un videojuego aun queda mucho trabajo por hacer y será necesario limar esas pequeñas asperezas especificas que puedan surgir durante su desarrollo. Aun así, el trabajo ha sido satisfactorio.

Actualmente la API BTR se esta utilizando para desarrollar juegos 2D de forma eficiente. Uno de ellos se utiliza como ayuda visual a otro TFG: *Desarrollo de un modulo de conexión entre móviles Android para comunicación en tiempo real en videojuegos multijugador*[1], por Javier Abella Taboada.

[1] http://ir.ii.uam.es/~fdiez/TFGs/propuestas/2013/201314_TFG_II_AlejandroSierra_3.pdf

10. Referencias

10.1. Libros

1. *Game and Graphics Programming for IOS and Android with OpenGL ES 2.0* - Romain Marucchi-Foino

2. *OpenGL ES 2.0 Programming Guide* - Aaftab Munshi, Dan Ginsburg, Dave Shreiner

3. *Desarrollo de Juegos Android* - Mario Zechner

4. *OpenGL ES 2 for Android: A Quick Start Guide* - Kevin Brothaler

10.2. Recursos en la Web

1. http://www.khronos.org/opengles/2_X/

2. http://www.opengl.org

3. http://developer.android.com/guide/topics/graphics/opengl.html

4. http://stackoverflow.com

5. http://gamedev.stackexchange.com

6. http://www.gamasutra.com

ANEXOS Técnicos

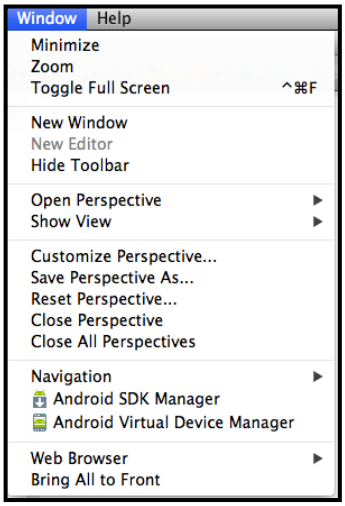
[1] MANUAL DE USUARIO

Instalación

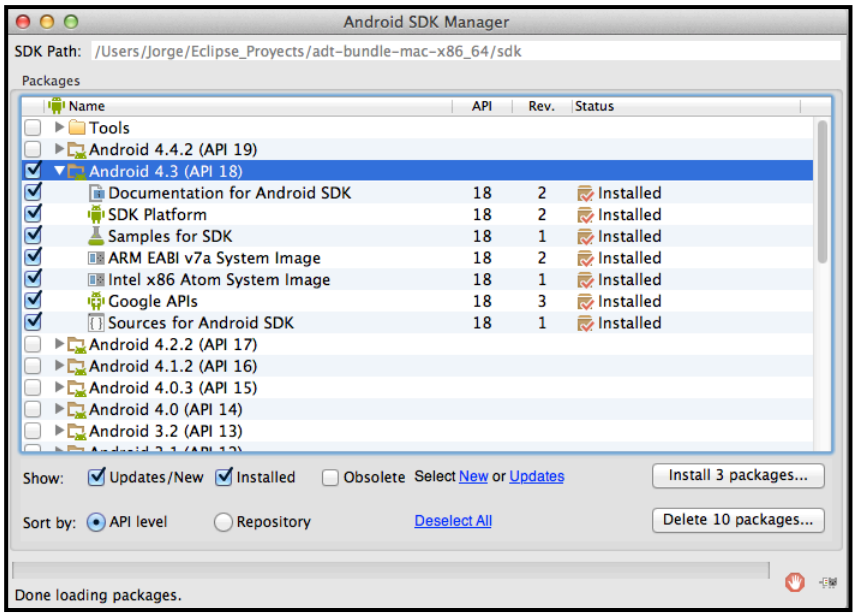
Antes de empezar es necesario descargar la IDE de eclipse con la SDK de android preparada. Esto se encuentra en la siguiente dirección:

http://developer.android.com/sdk/index.html?hl=sk

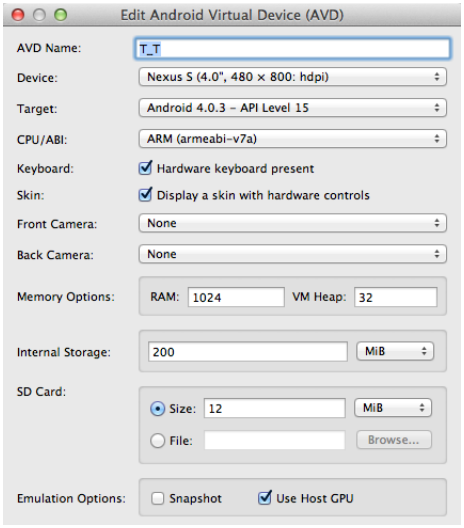
Una vez descargada es necesario actualizarla y preparar el AVD. Para ello se accede al gestor del SDK y al gestor del AVD:



Desde el gestor de la SDK es necesario bajarse la versión 18 para que funcione:



Desde el Gestor de AVD se prepara un emulador para su uso. Es importante saber que el emulador tiene fallos y que es posible que no funcione correctamente en algunos ordenadores. Lo mejor es utilizar un dispositivo móvil Android compatible. Aun así es posible que no exista ningún problema con el emulador.



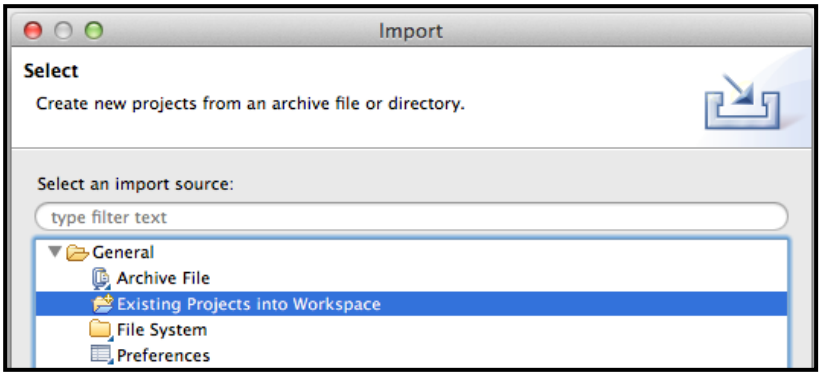
Esta configuración es la que se a utilizado durante todo el proyecto. Lo más importante es activar el uso de la GPU, sin la cual OpenGL no funcionaría, Use Host GPU.

El proyecto con la API BTR puede ser descargado de la siguiente dirección al servicio de almacenamiento Dropbox:

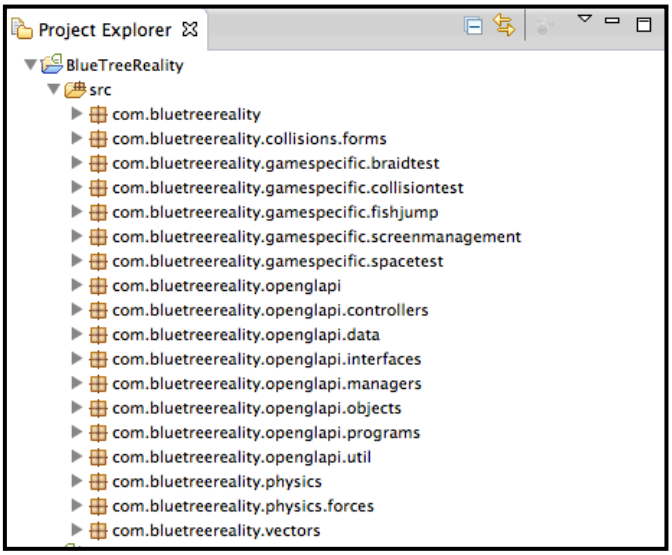
Código : https://www.dropbox.com/s/fx4bn0k2brv77os/TFG_Codigo.zip

Nota: En caso de no funcionar este enlace, solicitar el código en la siguiente dirección. jor.femenía@estudiante.uam.es.

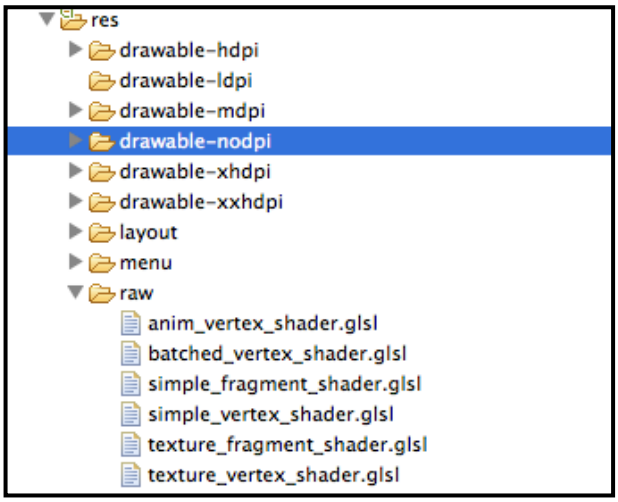
Se debe descargar el fichero y descomprimirlo. Se obtendrá así la carpeta con el nombre, *BlueTreeReality*. A continuación desde la IDE se debe abrir el proyecto:



Una vez cargado, el proyecto deberá aparecer de la siguiente forma en su explorador:



Ahí se encuentra todo el trabajo de creación de clases hecho durante el TFG. Y las dos otras carpetas importantes son *drawable-nodpi* y *raw*, donde se almacenan las imágenes y las shaders respectivamente:



También cabría mencionar el manifiesto de Android, si se estima oportuno se puede echar un vistazo y hacer las pertinentes modificaciones de orientación de pantalla, actualmente en modo *portrait.*

Uso

El uso de la API BTR es sencillo. En realidad el desarrollador solo tiene que ocuparse de 2 cosas, cargar sus recursos y hacer que interactuen en el tiempo de juego. Para ello, la API BTR ofrece todos los recursos mencionados en este documento.

El proyecto en si, viene con ejemplos preparados para que se pueda ver la forma de implementar las librerías. El desarrollador debería recurrir a ellos ya que están repletos de casos y formas diferentes de hacer uso de las posibilidades que existen. Estos ejemplos, que son los entornos de pruebas utilizados por el programador, están en el paquete *com.bluetreereality.gamespecific*.

Tal y como se ve en los ejemplos, el desarrollador debe de crear dos clases por cada pantalla que desee presentar. La clase desde donde se controlan los recursos, sea una subclase de *ScreenInterface* y la clase desde donde se crean, sea una subclase de *SpriteManagerResources*.

Estas clases tienen unos métodos abstractos que deberán rellenar. Estos métodos se vieron en la sección número 5 de esta memoria, aun así hagamos un repaso:

En la clase donde se crean los recursos deberán rellenar *resourcesToLoad* con los recursos a cargar, tal y como se ve en este ejemplo:

```
@Override
protected boolean resourcesToLoad(Context context) {
    spriteControllers.add(new SpriteController(CIRCLE_NAME,
            new SpriteObject(context, CIRCLE_SIZE, CIRCLE_SIZE, R.drawable.circle)));

    spriteControllers.add(new SpriteController(CIRCLE2_NAME,
            ((SpriteObject) this.getSpriteModel(CIRCLE_NAME)).makeCopy()));

    spriteControllers.add(new SpriteController(CRATE_NAME,
            new SpriteObject(context, CRATE_SIZE, CRATE_SIZE, R.drawable.crate)));

    auxSprite=(SpriteObject) this.getSpriteModel(CIRCLE2_NAME);
    auxSprite.setPosition(CIRCLE2_XPOS, CIRCLE2_YPOS);

    auxSprite=(SpriteObject) this.getSpriteModel(CRATE_NAME);
    auxSprite.setPosition(CRATE_XPOS, CRATE_YPOS);

    return true;
}
```

En está clase también se liberaran estos mismos recursos en el método *resourcesToFree*:

```
@Override
protected void resourcesToFree() {
    // TODO Auto-generated method stub
    ((SpriteObject) this.getSpriteModel(CIRCLE_NAME)).freeTexture();
    ((SpriteObject) this.getSpriteModel(CRATE_NAME)).freeTexture();

}
```

La clase que controla los recursos, *ScreenInterface*, es más extensa pero no por ello es más difícil de entender:

updateLogic: Aquí se producen todos los cálculos de la lógica del juego.

updateDraw: En este método se recogen los valores actualizados en la lógica y se expresa que debe ser pintado y como, o simplemente que cambios se han producido desde la última actualización.

onOnTouchEvent: Este método es el que responde a las entradas táctiles del usuario.

onOnDrawCreation: El método para la creación de sprites una vez se está ejecutando el juego.

Una complicación podrían ser los casting, puesto que el *SpriteManager* devuelve objetos genéricos. Es necesario que el desarrollador sepa que tipo de sprites añadió, para cuando los recupere para hacer modificaciones sobre ellos, pueda hacer los castings correctamente. Veamos un ejemplo:

```
spriteControllers.add(new SpriteController("Braid",
        new AnimatedSpriteObject(context,
                BRAID_PRINCESS_WIDTH_HEIGHT, BRAID_PRINCESS_WIDTH_HEIGHT,
                170f, 170f, R.drawable.braidchar, 6, 9, 49, 4)));
```

Así es como se carga un sprite animado. Se llama a *spriteControllers* dentro de *resourcesToLoad* y se le añade un nuevo controlador de sprite al que se le asigna un determinado nombre y un sprite, en este caso animado. Los parámetros están especificados:

com.bluetreereality.openglapi.objects.AnimatedSpriteObject.AnimatedSpriteObject(Context context, float quadWidth, float quadHeight, float spriteWidth, float spriteHeight, int textureId, int framesPerRow, int framesPerColumn, int totalFrames, int speed)

Builder

Parameters:
context Current Working Context
quadWidth Quad width size/2
quadHeight Quad height size/2
spriteWidth Texture Sprite Width(Portion taken form the texture)
spriteHeight Texture Sprite Height(Portion taken form the texture)
textureId ID(from R) to get the texture from
framesPerRow Frames per row(Frames MUST BE ALWAYS same-sized)
framesPerColumn Frames per Colum(Frames MUST BE ALWAYS same-sized)
totalFrames Total number of frames(Las row can have less frams that framesPerRow value)
speed Speed the frames will be drawn(max speed = 1 => 60 fps)

Una vez ha sido introducido lo volvemos a llamar de la siguiente manera:

```
auxAnimated= (AnimatedSpriteObject) spriteManager.getSpriteModel("Braid");
auxAnimated.setV_flipped(true);
if(running){
    auxAnimated.setSubAnimation("RUN");
    auxAnimated.setScale(1.5f);
}
auxAnimated.setPosition(x_pos_char1, y_pos_char1);
```

Se puede observar como es llamado desde *updateSprites*, obteniéndolo del *spriteManager*, propio de este método, y siendo almacenado en una variable haciendo el correspondiente casting al tipo de objeto. Luego, se hacen las modificaciones oportunas, en este ejemplo se voltea verticalmente y si el personaje está corriendo se cambia la animación y se escala. También se varía la posición.

Para liberar dicho sprite:

```
((AnimatedSpriteObject) this.getSpriteModel("Braid")).freeTexture();
```

Esta vez, se obtiene de la propia clase, ya que volvemos a estar en una extensión de *SpriteManagerResources*, concretamente en el método *resourcesToFree*.

Todo lo visto es una sola pantalla, para cambiar de pantalla lo que hay que hacer es decir cual será la siguiente pantalla por la cual vas a sustituir a la actual y marcarla como acabada. Para ello se utilizan los métodos *this.finish* y *this.setNextScreen*. Vemos ahora como se introducen pantallas para poder llamarlas, esto es lo único que hay que modificar en el main:

```
//Screens can be added here or later//But you have to set at least 1 to display
screenManager= new ScreenManager();
screenManager.addScreen(new ScreenController("Menu", new SpriteManager(new GameMenuResources()), new GameMenu(screenManager)));
screenManager.setCurrentScreen("Menu");
```

En este caso especifico se pasa al *GameMenu*, la pantalla, el propio *ScreenManager* ya que la pantalla representa un menú de pantallas donde se selecciona un juego y se inserta la pantalla que se va a utilizar. También se podrían cargar todas las pantallas directamente y llamarlas solo para el cambio, según se maneje mejor el programador.

Eso es todo. Se recomienda mirar los ejemplos y hacer pruebas con ellos para llegar a un completo entendimiento. El proceso es simple, cargar sprite, referenciar sprite y modificar sprite. Todo en sus debidos tiempos.

[2] Shaders Implementadas

Todas los programas utilizados hacen uso de texturas, por lo tanto la shader de fragmentos es común a todas ellas y es la siguiente:

```
precision mediump float;

uniform sampler2D u_TextureUnit;

varying vec2 v_TextureCoordinates;

void main(){
     gl_FragColor= texture2D(u_TextureUnit, v_TextureCoordinates);
}
```

La única función que realiza esta shader es definir a raíz de las coordenadas de la textura recibidas, el color final del fragmento gestionado.

A continuación se ven las shaders de vértices:

Única Textura(Sprite Normal)

Es la shader de vértices normal para un objeto al que se le va aplicar una textura. Resuelve la posición final de un vector y de las coordenadas de una textura.

```
uniform mat4 u_Matrix;

attribute vec4 a_Position;
attribute vec2 a_TextureCoordinates;

varying vec2 v_TextureCoordinates;

void main(){
     v_TextureCoordinates = a_TextureCoordinates;
     gl_Position = u_Matrix * a_Position;
}
```

Animada(Sprite Animado)

Esta shader es más compleja que la anterior ya que debe resolver la posición dentro de la textura donde está la porción a representar. Para ello a parte de moverse por los ejes, la shader también recibe un dato *Displacement* para realizar la conversión del tamaño de la textura con la que trabaja el desarrollador al tamaño con el que trabaja OpenGL que es de 0 a 1 siempre. Este se calcula antes de ser referenciado:

```
uniform mat4 u_Matrix;

uniform float u_XSpritePos;
```

```
uniform float u_YSpritePos;
uniform float u_XDisplacement;
uniform float u_YDisplacement;

attribute vec4 a_Position;
attribute vec2 a_TextureCoordinates;

varying vec2 v_TextureCoordinates;

void main(){
     v_TextureCoordinates.x= (a_TextureCoordinates.x + u_XSpritePos) *
u_XDisplacement;
     v_TextureCoordinates.y= (a_TextureCoordinates.y + u_YSpritePos) *
u_YDisplacement;

     gl_Position = u_Matrix * a_Position;
}
```

De Grupo(Sprite Batched)

Esta última shader de vértices sigue el mismo patrón que la anterior pero esta vez se mueve libremente por la textura. Es decir, realiza la conversión de forma diferente para permitir desplazamientos personalizados. En realidad con esta versión se podría elaborar la anterior también:

```
uniform mat4 u_Matrix;

uniform float u_XOffset;
uniform float u_YOffset;
uniform float u_XDisplacement;
uniform float u_YDisplacement;

attribute vec4 a_Position;
attribute vec2 a_TextureCoordinates;

varying vec2 v_TextureCoordinates;

void main(){
     v_TextureCoordinates.x= (a_TextureCoordinates.x * u_XDisplacement)
+ u_XOffset;
     v_TextureCoordinates.y= (a_TextureCoordinates.y * u_YDisplacement)
+ u_YOffset;

     gl_Position = u_Matrix * a_Position;
}
```